Pensar en C++ *(Volumen 2)*

Traducción (INACABADA) del libro Thinking in C++, Volumen 2

Bruce Eckel

12 de enero de 2012

Puede encontrar la versión actualizada de este libro e información adicional sobre el proyecto de traducción en
http://arco.esi.uclm.es/~david.villa/pensarC++.html

Pensar en C++ ***(Volumen 2)***
by Bruce Eckel

Copyright © 2004 Bruce Eckel

Índice general

1. Introducción **1**
1.1. Objetivos 1
1.2. Capítulos 2
1.3. Ejercicios 3
1.3.1. Soluciones de los ejercicios 4
1.4. Código fuente 4
1.5. Compiladores 4
1.6. Estándares del lenguaje 5
1.7. Seminarios, CD-ROMs y consultoría 5
1.8. Errores 6
1.9. Sobre la portada 6
1.10. Agradecimientos 6

I Construcción de Sistemas estables **9**

2. Tratamiento de excepciones **13**
2.1. Tratamiento tradicional de errores 13
2.2. Lanzar una excepción 15
2.3. Capturar una excepción 15
2.3.1. El bloque `try` 15
2.3.2. Manejadores de excepción 16
2.3.3. 16
2.4. 16
2.4.1. Capturar cualquier excepción 17
2.4.2. Relanzar una excepción 18
2.4.3. Excepciones no capturadas 18
2.5. Limpieza 18
2.5.1. Gestión de recursos 19
2.5.2. 20

2.5.3. `auto_ptr` 22
2.5.4. Bloques `try` a nivel de función 22
2.6. Excepciones estándar 23
2.7. Especificaciones de excepciones 24
2.7.1. ¿Mejores especificaciones de excepciones? 25
2.7.2. Especificación de excepciones y herencia 26
2.7.3. Cuándo no usar especificaciones de excepción 26
2.8. Seguridad de la excepción 26
2.9. Programar con excepciones 28
2.9.1. Cuándo evitar las excepciones 28
2.9.2. Usos típicos de excepciones 28
2.10. Sobrecarga 28
2.11. Resumen 29
2.12. Ejercicios 29

3. Programación defensiva **31**
3.1. Aserciones 33
3.2. Un framework de pruebas unitarias sencillo 35
3.2.1. Pruebas automatizadas 37
3.2.2. El Framework TestSuite 39
3.2.3. Suites de test 42
3.2.4. El código del framework de prueba 43
3.3. Técnicas de depuración 48
3.3.1. Macros de seguimiento 48
3.3.2. Fichero de rastro 49
3.3.3. Encontrar agujeros en memoria 50
3.4. Resumen 55
3.5. Ejercicios 55

II La librería Estándar de C++ **57**

4. Las cadenas a fondo **61**
4.1. ¿Qué es un `string`? 61
4.2. Operaciones con cadenas 63
4.2.1. Añadiendo, insertando y concatenando cadenas 63
4.2.2. Reemplazar caracteres en cadenas 65
4.2.3. Concatenación usando operadores no-miembro sobrecargados 68
4.3. Buscar en cadenas 69

4.3.1. Busqueda inversa 73
4.3.2. Encontrar el primero/último de un conjunto de caracteres . . . 74
4.3.3. Borrar caracteres de cadenas 76
4.3.4. Comparar cadenas 78
4.3.5. Cadenas y rasgos de caracteres 82
4.4. Una aplicación con cadenas 86
4.5. Resumen 90
4.6. Ejercicios 91

5. Iostreams **95**
5.1. ¿Por que `iostream`? 95
5.2. `Iostreams` al rescate 99
5.2.1. Insertadores y extractores 99
5.2.2. Uso común 102
5.2.3. Entrada orientada a líneas 104
5.2.3.1. Versiones sobrecargadas de `get()` 105
5.2.3.2. Leyendo bytes sin formato 105
5.3. Manejo errores de `stream` 105
5.3.1. Estados del `stream` 105
5.3.2. Streams y excepciones 107
5.4. Iostreams de fichero 107
5.4.1. Un ejemplo de procesado de fichero. 108
5.4.2. Modos de apertura 109
5.5. Almacenamiento de `iostream` 110
5.6. Buscar en iostreams 112
5.7. Iostreams de `string` 115
5.7.1. Streams de cadena de entrada 115
5.7.2. Streams de cadena de salida 117
5.8. Formateo de stream de salida 119
5.8.1. Banderas de formateo 120
5.8.2. Campos de formateo 121
5.8.3. Anchura, relleno y precisión 122
5.8.4. Un ejemplo exhaustivo 123
5.9. Manipuladores 126
5.9.1. Manipuladores con argumentos 127
5.9.2. 129
5.9.3. 130
5.10. 131

5.10.1. 131
5.10.2. 133
5.10.3. 135
5.11. 138
5.11.1. 138
5.11.2. 138
5.12. 139
5.13. 139

6. Las plantillas en profundidad **141**
6.1. 141
6.1.1. 141
6.1.2. 142
6.1.3. 143
6.1.4. 146
6.1.4.1. 148
6.1.4.2. 148
6.1.5. 148
6.1.6. 149
6.2. 150
6.2.1. 150
6.2.2. 152
6.2.3. 153
6.2.4. 154
6.2.5. 156
6.3. 157
6.3.1. 157
6.3.2. 158
6.3.3. 159
6.3.4. 161
6.4. 164
6.4.1. 164
6.4.2. 166
6.5. 169
6.5.1. 169
6.5.2. 171
6.5.3. 173
6.6. 174

6.6.1. . . . 175
6.6.1.1. . . . 176
6.6.1.2. . . . 177
6.6.1.3. . . . 178
6.6.2. . . . 180
6.7. . . . 183
6.7.1. . . . 183
6.7.2. . . . 183
6.7.3. . . . 184
6.7.4. . . . 184
6.8. . . . 184

7. Algoritmos genéricos **187**
7.1. Un primer vistazo . . . 187
7.1.1. Predicados . . . 190
7.1.2. Iteradores de flujo . . . 192
7.1.3. Complejidad algorítmica . . . 194
7.2. Objetos-función . . . 194
7.2.1. Clasificación de objetos-función . . . 195
7.2.2. Creación automática de objetos-función . . . 195
7.2.3. Objetos-función adaptables . . . 196
7.2.4. Más ejemplos de objetos-función . . . 196
7.2.5. Adaptadores de puntero a función . . . 199
7.2.6. Escribir sus propios adaptadores de objeto-función . . . 202
7.3. Un catálogo de algoritmos STL . . . 205
7.3.1. Herramientas de soporte para la creación de ejemplos . . . 205
7.3.1.1. Reordenación estable vs. inestable . . . 205
7.3.2. Relleno y generación . . . 206
7.3.2.1. Ejemplo . . . 206
7.3.3. Conteo . . . 207
7.3.3.1. Ejemplo . . . 207
7.3.4. Manipulación de secuencias . . . 208
7.3.4.1. Ejemplo . . . 208
7.3.5. Búsqueda y reemplazo . . . 209
7.3.5.1. Ejemplo . . . 209
7.3.6. Comparación de rangos . . . 211
7.3.6.1. Ejemplo . . . 211
7.3.7. Eliminación de elementos . . . 212

7.3.7.1. Ejemplo 212
7.3.8. Ordenación y operación sobre rangos ordenados 213
7.3.8.1. Ordenación 213
7.3.8.2. Ejemplo 213
7.3.8.3. Mezcla de rangos ordenados 214
7.3.8.4. Ejemplo 214
7.3.8.5. Ejemplo 215
7.3.9. Operaciones sobre el montículo 215
7.3.10. Aplicando una operación a cada elemento de un rango 215
7.3.10.1. Ejemplos 215
7.3.11. Algoritmos numéricos 219
7.3.11.1. Ejemplo 219
7.3.12. Utilidades generales 220
7.4. Creando sus propios algoritmos tipo STL 220
7.5. Resumen 221
7.6. Ejercicios 221

III Temas especiales 223

8. Herencia múltiple 227
8.1. Perspectiva 227
8.2. Herencia de interfaces 228
8.3. Herencia de implementación 231
8.4. Subobjetos duplicados 234
8.5. Clases base virtuales 236
8.6. Cuestión sobre búsqueda de nombres 240
8.7. Evitar la MI 242
8.8. Extender una interface 242
8.9. Resumen 245
8.10. Ejercicios 245

9. Patrones de Diseño 247
9.1. El Concepto de Patrón 247
9.1.1. La composición es preferible a la herencia 248
9.2. Clasificación de los patrones 248
9.2.1. Características, modismos patrones 249
9.3. Simplificación de modismos 249
9.3.1. Mensajero 250

9.3.2. Parámetro de Recolección 251
9.4. Singleton 252
9.4.1. Variantes del Singleton 253
9.5. Comando: elegir la operación 256
9.5.1. Desacoplar la gestión de eventos con Comando 258
9.6. Desacoplamiento de objetos 261
9.6.1. Proxy: FIXME: hablando en nombre de otro objeto 261
9.6.2. Estado: cambiar el comportamiento del objeto 262
9.7. Adaptador 264
9.8. Template Method 266
9.9. Estrategia: elegir el algoritno en tiempo de ejecución 267
9.10. Cadena de Responsabilidad: intentar una secuencia de estrategias . . . 268
9.11. Factorías: encapsular la creación de objetos 270
9.11.1. Factorías polimórficas 272
9.11.2. Factorías abstractas 275
9.11.3. Constructores virtuales 277
9.12. Builder: creación de objetos complejos 279
9.13. Observador 284
9.13.1. El ejemplo de observador 286
9.14. Despachado múltiple 289
9.14.1. Despachado múltiple con Visitor 291
9.15. Resumen 293
9.16. Ejercicios 293

10. Concurrencia **295**
10.1. Motivación 296
10.2. Concurrencia en C++ 297
10.2.1. Instalación de ZThreads 297
10.2.2. Definición de tareas 298
10.3. Utilización de los hilos 300
10.3.1. Creación de interfaces de usuarios interactivas 301
10.3.2. Simplificación con Ejecutores 303
10.3.3. Ceder el paso 306
10.3.4. Dormido 307
10.3.5. Prioridad 308
10.4. Comparición de recursos limitados 310
10.4.1. Aseguramiento de la existencia de objetos 310
10.4.2. Acceso no apropiado a recursos 313

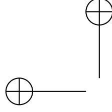
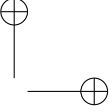
10.4.3. Control de acceso 315
10.4.4. Código simplificado mediante guardas 317
10.4.5. Almacenamiento local al hilo 320
10.5. Finalización de tareas 321
10.5.1. Prevención de coliciones en iostream 322
10.5.2. El jardín ornamental 323
10.5.2.1. Operaciones atómicas 326
10.5.3. FIXME:Teminación al bloquear 326
10.5.4. Interrupción 327
10.6. Cooperación entre hilos 331
10.6.1. Wait y signal 331
10.6.2. Relación de productor/consumidor 332
10.6.3. Resolución de problemas de hilos mediante colas 335
10.6.4. Broadcast 339
10.7. Bloqueo letal 343
10.8. Resumen 346
10.9. Ejercicios 346

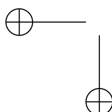
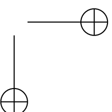
Índice de figuras

8. Herencia múltiple

8.1. 228

8.2. 228

1: Introducción

En el volumen 1 de este libro, aprendió los fundamentos de C y C++. En este volumen, veremos características más avanzadas, con miras hacia técnicas e ideas para realizar programas robustos en C++.

Asumimos que está familiarizado con el material presentado en el Volumen 1.

1.1. Objetivos

Nuestros objetivos en este libros son:

1. Presentar el material como un sencillo paso cada vez, de este modo el lector puede asimilar cada concepto antes de seguir adelante.

2. Enseñar técnicas de "programación práctica" que puede usar en la base del día a día.

3. Darle lo que pensamos que es importante para que entienda el lenguaje, más bien que todo lo que sabemos. Creemos que hay una "jerarquía de importancia de la información", y hay algunos hechos que el 95% de los programadores nunca necesitarán conocer, sino que sólo confundiría a la gente y añade complejidad a su percepción del lenguaje. Tomando un ejemplo de C, si memoriza la tabla de prioridad de operadores (nosotros nunca lo hicimos) puede escribir código ingenioso. Pero si debe pensarlo, confundirá al lector/mantenedor de ese código. De modo que olvide las precedencias y use paréntesis cuando las cosas no estén claras.

4. Manténgase suficientemente centrado en cada sección de modo que el tiempo de lectura -y el tiempo entre ejercicios- sea pequeño. Esto no sólo hace mantener las mentes de los lectores más activas e involucradas durante un seminario práctico sino que da al lector un mayor sentido de éxito.

Hemos procurado no usar ninguna versión de un vendedor particular de C++. Hemos probado el código sobre todas las implementaciones que pudimos (descriptas posteriormente en esta introducción), y cuando una implementación no funciona en absoluto porque no cumple el Estándar C++, lo hemos señalado en el ejemplo(verá las marcas en el código fuente) para excluirlo del proceso de construcción.

6. Automatizar la compilación y las pruebas del código en el libro. Hemos descubierto que el código que no está compilado y probado probablemente no funcione correctamente, de este modo en este volumen hemos provisto a los ejemplos con código de pruebas. Además, el código que puede descargar desde http://www.MindView.net ha sido extraído directamente del texto del libro usando programas que automáticamente crean makefiles para compilar y ejecutar las pruebas. De esta forma sabemos que el código en el libro es correcto.

1.2. Capítulos

Aquí está una breve descripción de los capítulos que contiene este libro:

Parte 1: Construcción de sistemas estables

1. Manejar excepciones. Manejar errores ha sido siempre un problema en programación. Incluso si obedientemente devuelve información del error o pone una bandera, la función que llama puede simplemente ignorarlo. Manejar excepciones es una cualidad primordial en C++ que soluciona este problema permitiéndole "lanzar" un objeto fuera de su función cuando ocurre un error crítico. Tire diferentes tipos de objetos para diferentes errores, y la función que llama "coge" estos objetos en rutinas separadas de gestión de errores. Si lanza una excepción, no puede ser ignorada, de modo que puede garantizar que algo ocurrirá en respuesta a su error. La decisión de usar excepciones afecta al diseño del código positivamente, de modo fundamental.

2. Programación defensiva. Muchos problemas de software pueden ser prevenidos. Programar de forma defensiva es realizar cuidadosamente código de tal modo que los bugs son encontrados y arreglados pronto antes que puedan dañar en el campo. Usar aserciones es la única y más importante forma para validar su código durante el desarrollo, dejando al mismo tiempo seguimiento ejecutable de la documentación en su código que muestra sus pensamientos mientras escribe el código en primer lugar. Pruebe rigurosamente su código antes de darlo a otros. Un marco de trabajo de pruebas unitario automatizado es una herramienta indispensable para el éxito, en el desarrollo diario de software.

Parte 2: La biblioteca estándar de C++

3. Cadenas en profundidad. La actividad más común de programación es el procesamiento de texto. La clase string de C++ libera al programador de los temas de gestión de memoria, mientras al mismo tiempo proporciona una fuente de recursos para el procesamiento de texto. C++ también facilita el uso de una gran variedad de caracteres y locales para las aplicaciones de internacionalización.

4. Iostreams. Una de las bibliotecas original de C++-la que proporciona la facilidad básica de I/O-es llamada iostreams. Iostreams está destinado a reemplazar stdio.h de C con una biblioteca I/O que es más fácil de usar, más flexible, y extensible-que puede adaptarla para trabajar con sus nuevas clases. Este capítulo le enseña cómo hacer el mejor uso de la biblioteca actual I/O, fichero I/O, y formateo en memoria.

5. Plantillas en profundidad. La distintiva cualidad del "C++ moderno" es el extenso poder de las plantillas. Las plantillas hacen algo más que sólo crear contenedores genéricos. Sostienen el desarrollo de bibliotecas robustas, genéricas y de alto rendimiento.Hay mucho por saber sobre plantillas-constituyen, como fue, un sublenguaje dentro del lenguaje C++, y da al programador un grado impresionante de control sobre el proceso de compilación. No es una exageración decir que las plantillas han revolucionado la programación de C++.

6. Algoritmos genéricos. Los algoritmos son el corazón de la informática, y C++, por medio de la facilidad de las plantillas, facilita un impresionante entorno de poder, eficiencia, y facilidad de uso de algoritmos genéricos. Los algoritmos estándar son también personalizables a través de objetos de función. Este capítulo examina cada algoritmo de la biblioteca. (Capítulos 6 y 7 abarcan esa parte de la biblioteca Estándar de C++ comúnmente conocida como Biblioteca de Plantilla Estándar, o STL.)

7. Contenedores genéricos e Iteradores. C++ proporciona todas las estructuras comunes de datos de modo de tipado fuerte. Nunca necesita preocuparse sobre qué tiene tal contenedor. La homogenidad de sus objetos está garantizada. Separar la

#FIXME traversing de un contenedor del propio contenedor, otra realización de plantillas, se hace posible por medio de los iteradores. Este ingenioso arreglo permite una aplicación flexible de algoritmos a contenedores usando el más sencillo de los diseños.

Parte 3: Temas especiales

8. Identificación de tipo en tiempo de ejecución. La identificación de tipo en tiempo de ejecución (RTTI) encuentra el tipo exacto de un objeto cuando sólo tiene un puntero o referencia al tipo base. Normalmente, tendrá que ignorar a propósito el tipo exacto de un objeto y permitir al mecanismo de función virtual implementar el comportamiento correcto para ese tipo. Pero ocasionalmente (como las herramientas de escritura de software tales como los depuradores) es útil para conocer el tipo exacto de un objeto-con su información, puede realizar con frecuencia una operación en casos especiales de forma más eficiente. Este capítulo explica para qué es RTTT y como usarlo.

9. Herencia múltiple. Parece sencillo al principio: Una nueva clase hereda de más de una clase existente. Sin embargo, puede terminar con copias ambiguas y múltiples de objetos de la clase base. Ese problema está resuelto con clases bases virtuales, pero la mayor cuestión continua: ¿Cuándo usarla? La herencia múltiple es sólo imprescindible cuando necesite manipular un objeto por medio de más de un clase base común. Este capítulo explica la sintaxis para la herencia múltiple y muestra enfoques alternativos- en particular, como las plantillas solucionan un problema típico. Usar herencia múltiple para reparar un interfaz de clase "dañada" demuestra un uso valioso de esta cualidad.

10. Patrones de diseño. El más revolucionario avance en programación desde los objetos es la introducción de los patrones de diseño. Un patrón de diseño es una codificación independiente del lenguaje de una solución a un problema de programación común, expresado de tal modo que puede aplicarse a muchos contextos. Los patrones tales como Singleton, Factory Method, y Visitor ahora tienen lugar en discusiones diarias alrededor del teclado. Este capítulo muestra como implementar y usar algunos de los patrones de diseño más usados en C++.

11. Programación concurrente. La gente ha llegado a esperar interfaces de usuario sensibles que (parece que) procesan múltiples tareas simultáneamente. Los sistemas operativos modernos permiten a los procesos tener múltiples hilos que comparten el espacio de dirección del proceso. La programación multihilo requiere una perspectiva diferente, sin embargo, y viene con su propio conjunto de dificultades. Este capítulo utiliza un biblioteca disponible gratuitamente (la biblioteca ZThread por Eric Crahen de IBM) para mostrar como gestionar eficazmente aplicaciones multihilo en C++.

1.3. Ejercicios

Hemos descubierto que los ejercicios sencillos son excepcionalmente útiles durante un seminario para completar la comprensión de un estudiante. Encontrará una colección al final de cada capítulo.

Estos son bastante sencillos, de modo que puedan ser acabados en una suma de tiempo razonable en una situación de clase mientras el profesor observa, asegurándose que todos los estudiantes están absorbiendo el material. Algunos ejercicios son un poco más exigentes para mantener entretenidos a los estudiantes avanzados. Están todos diseñados para ser resueltos en un tiempo corto y sólo están allí para probar y refinar su conocimiento más bien que presentar retos mayores (presumible-

mente, podrá encontrarlos o más probablemente ellos le encontrarán a usted).

1.3.1. Soluciones de los ejercicios

Las soluciones a los ejercicios pueden encontrarse en el documento electrónico La Guía de Soluciones Comentada de C++, Volumen 2, disponible por una cuota simbólica en http://www.MindView.net.

1.4. Código fuente

El código fuente para este libro está autorizado como software gratuito, distribuido por medio del sitio web http://www.MindView.net. Los derechos de autor le impiden volver a publicar el código impreso sin permiso.

En el directorio inicial donde desempaqueta el código encontrará el siguiente aviso de derechos de autor:

Puede usar el código en sus proyectos y en clase siempre que el aviso de los derechos de autor se conserve.

1.5. Compiladores

Su compilador podría no soportar todas las cualidades discutidas en este libro, especialmente si no tiene la versión más nueva de su compilador. Implementar un lenguaje como C++ es un tarea Hercúlea, y puede suponer que las cualidades aparecen por partes en lugar de todas juntas. Pero si intenta uno de los ejemplos del libro y obtiene muchos errores del compilador, no es necesariamente un error en el código o en el compilador-puede que sencillamente no esté implementado todavía en su compilador concreto.

Empleamos un número de compiladores para probar el código de este libro, en un intento para asegurar que nuestro código cumple el Estándar C++ y funcionará con todos los compiladores posibles. Desafortunadamente, no todos los compiladores cumplen el Estándar C++, y de este modo tenemos un modo de excluir ciertos ficheros de la construcción con esos compiladores. Estas exclusiones se reflejadas automáticamente en los makefiles creados para el paquete de código para este libro que puede descargar desde www.MindView.net. Puede ver las etiquetas de exclusión incrustadas en los comentarios al inicio de cada listado, de este modo sabrá si exigir a un compilador concreto que funcione con ese código (en pocos casos, el compilador realmente compilará el código pero el comportamiento de ejecución es erróneo, y excluimos esos también).

Aquí están las etiquetas y los compiladores que se excluyen de la construcción.

· {-dmc} El Compilador de Mars Digital de Walter Bright para Windows, descargable gratuitamente en www.DigitalMars.com. Este compilador es muy tolerante y así no verá casi ninguna de estas etiquetas en todo el libro.

· {-g++} La versión libre Gnu C++ 3.3.1, que viene preinstalada en la mayoría de los paquetes Linux y Macintosh OSX. También es parte de Cygwin para Windows (ver abajo). Está disponible para la mayoría de las plataformas en gcc.gnu.org.

· {-msc} Microsoft Version 7 con Visual C++ .NET (viene sólo con Visual Studio .NET; no descargable gratuitamente).

· {-bor} Borland C++ Version 6 (no la versión gratuita; éste está más actualizado).

· {-edg} Edison Design Group (EDG) C++. Este es el compilador de referencia para la conformidad con los estándares. Esta etiqueta existe a causa de los temas de biblioteca, y porque estábamos usando un copia gratis de la interfaz EDG con una implementación de la biblioteca gratuita de Dinkumware, Ltd. No aparecieron errores de compilación a causa sólo del compilador.

· {-mwcc} Metrowerks Code Warrior para Macintosh OS X. Fíjese que OS X viene con Gnu C++ preinstalado, también.

Si descarga y desempaqueta el paquete de código de este libro de www.MindView.net, encontrará los makefiles para construir el código para los compiladores de más arriba. Usabamos GNU-make disponible gratuitamente, que viene con Linux, Cygwin (una consola gratis de Unix que corre encima de Windows; ver www.Cygwin.com), o puede instalar en su plataforma, ver www.gnu.org/software/make. (Otros makes pueden o no funcionar con estos ficheros, pero no están soportados.) Una vez que instale make, si teclea make en la línea de comando obtendrá instrucciones de cómo construir el código del libro para los compiladores de más arriba.

Fíjese que la colocación de estas etiquetas en los ficheros en este libro indica el estado de la versión concreta del compilador en el momento que lo probamos. Es posible y probable que el vendedor del compilador haya mejorado el compilador desde la publicación de este libro. Es posible también que mientras que realizamos el libro con tantos compiladores, hayamos desconfigurado un compilador en concreto que en otro caso habría compilado el código correctamente. Por consiguiente, debería probar el código usted mismo con su compilador, y comprobar también el código descargado de www.MindView.net para ver que es actual.

1.6. Estándares del lenguaje

A lo largo de este libro, cuando se hace referencia a la conformidad con el ANSI/ISO C estándar, estaremos haciendo referencia al estándar de 1989, y de forma general diremos solamente 'C.'. Sólo si es necesario distinguir entre el Estándar de C de 1989 y anteriores, versiones pre-Estándares de C haremos la distinción. No hacemos referencia a c99 en este libro.

El Comité de ANSI/ISO C++ hace mucho acabó el trabajo sobre el primer Estándar C++, comunmente conocido como C++98. Usaremos el término Standard C++ para referirnos a este lenguaje normalizado. Si nos referimos sencillamente a C++, asuma que queremos decir "Standard C++". El Comité de Estándares de C++ continua dirigiendo cuestiones importantes para la comunidad de C++ que se convertirá en C++0x, un futuro Estándar de C++ que no estará probablemente disponible durante muchos años.

1.7. Seminarios, CD-ROMs y consultoría

La compañía de Bruce Eckel, MindView, Inc., proporciona seminarios públicos prácticos de formación basados en el material de este libro, y también para temas avanzados. El material seleccionado de cada capítulo representa una lección, que es sucedida por un periodo de ejercicios guiado de tal modo que cada estudiante recibe atención personalizada. También facilitamos formación en el lugar, consultoría, tutoría y comprobación de diseño y código. La información y los formularios de inscripción para los seminarios próximos y otra información de contacto se puede

encontrar en http://www.MindView.net.

1.8. Errores

No importa cuantos trucos usen los escritores para detectar errores, algunos siempre pasan desapercibidos y éstos a menudo destacan para un nuevo lector. Si descubre algo que cree ser un error, por favor use el sistema de respuesta incorporado en la versión electrónica de este libro, que encontrará en http://www.MindView.net. Su ayuda se valora.

1.9. Sobre la portada

El arte de la portada fue realizado por la mujer de Larry O'Brien, Tina Jensen (sí, el Larry O'Brien quien fue el editor de Software Development Magazine durante muchos años). No solamente los dibujos son bonitos, son también sugerencias excelentes de polimorfismo. La idea de usar estas imágenes viene de Daniel Will-Harris, el diseñador de portadas (www.Will-Harris.com), trabajando con Bruce.

1.10. Agradecimientos

El volumen 2 de este libro se descuidó a mitad de estar acabado durante mucho tiempo mientras Bruce se distraía con otras cosas, en particular con Java, Patrones de Diseño y especialmente con Python (ver www.Python.org). Si Chuck no hubiera estado dispuesto (tontamente, él reflexiona algunas veces) a acabar la otra mitad y llevar las cosas al día, este libro casi seguramente no habría existido. No hay mucha gente a quien Bruce habría confiado tranquilamente este libro. La afición de Chuck por la precisión, la corrección y la explicación clara es lo que ha hecho que este libro sea tan bueno como es.

Jamie King trabajó como persona de prácticas bajo la dirección de Chuck durante la finalización de este libro. El fue una parte esencial en asegurarse que el libro se finalizaba, no sólo proporcionando contestación a Chuck, sino especialmente por su interrogatorio implacable y siendo puntilloso con cada elección que él no comprendía por completo. Si sus preguntas son respondidas por este libro, es probablemente porque Jamie las preguntó primero. Jamie también mejoró unos cuantos programas de ejemplo y creó muchos de los ejercicios al final de cada capítulo. Scott Baker, otro trabajador en prácticas de Chuck patrocinado por MindView, Inc., ayudó con los ejercicios del capítulo 3.

Eric Crahen de IBM fue decisivo en la finalización del capítulo 11 (Concurrencia). Cuando estábamos buscando un paquete de hilos, buscamos uno que fuese intuitivo y fácil de usar, mientras fuese suficientemente robusto para hacer el trabajo. Con Eric conseguimos esto y posteriormente- él estuvo sumamente cooperativo y ha usado nuestras contestaciones para mejorar su biblioteca, mientras nosotros también nos hemos beneficiado de su conocimiento.

Estamos agradecidos a Pete Becker por ser nuestro editor técnico. Pocas personas son tan elocuentes y exigentes como Pete, ni mencionar como experto en C++ y desarrollo de software en general. También dar gracias a Bjorn Karlsson por su cortés y oportuna asistencia técnica revisando el manuscrito entero con escaso aviso.

Walter Bright hizo esfuerzos Hercúleos para asegurarse que su compilador Digital Mars C++ compilaría los ejemplos de este libro. Puso disponible el compilador

mediante descarga gratuita en http://www.DigitalMars.com. ¡Gracias, Walter!

Las ideas y conocimientos de este libro provienen también de muchas otras fuentes: amigos como Andrea Provaglio, Dan Saks, Scott Meyers, Charles Petzold, y Michael Wilk; los pioneros del lenguaje como Bjarne Stroustrup, Andrew Koening, y Rob Murray; los miembros del Comité de Estándares de C++ como Nathan Myers (quien fue especialmente servicial y generoso con sus perspicacias), Herb Sutter, PJ Plauger, Kevlin Henney, David Abrahams, Tom Plum, Reg Charney, Tom Penello, Sam Druker, Uwe Steinmueller, John Spicer, Steve Adamczyk, y Daveed Vandevoorde; la gente que ha hablado en el apartado de C++ de la Conferencia de Desarrollo de Software (que Bruce creó y desarrolló, y en la que habló Chuck); Compañeros de Chuck como Michael Seaver, Huston Franklin, David Wagstaff, y muchos estudiantes en seminarios, quienes realizaron las preguntas que necesitamos escuchar para hacer el material más claro.

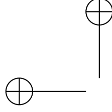
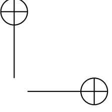
El diseño del libro, la elección de la fuente, el diseño de la portada, y la foto de la portada ha sido realizada por el amigo de Bruce Daniel Will-Harris, célebre autor y desiñador, que solía jugar con cartas temporales en el primer ciclo de secundaria mientras esperaba la invención de los ordenadores y la publicación asistida por ordenador. Sin embargo, presentamos las páginas listas para imprimir nosotros mismos, por lo tanto los errores de composición tipográfica son nuestros. Se ha usado Microsoft® Word XP para escribir el libro y crear la versión lista para imprimir. El cuerpo del texto está en Verdana y los títulos está en Verdana. El tipo de letra del código es Courier New.

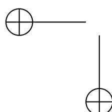
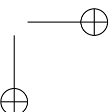
Deseamos también dar las gracias a los múltiples profesionales en el Edison Design Group y Dinkumware, Ltd., por darnos copias gratis de su compilador y biblioteca (respectivamente). Sin su experta asistencia, dada gentilmente, algunos de los ejemplos de este libro no podrían haber sido probados- También queremos agradecer a Howard Hinnant y a la gente de Metrowerks por la copia de su compilador, y a Sandy Smith y la gente de SlickEdit por facilitar a Chuck un entorno de edición durante muchos años. Greg Comeau también facilitó un copia de su exitoso compilador basado en EDG, Comeau C++.

Gracias especialmente a todos nuestros profesores, y a todos nuestros estudiantes (que son también nuestros profesores).

Evan Cofsky (Evan@TheUnixMan.com) facilitó todo tipo de asistencia en el servidor también con el desarrollo de programas en su ahora lenguaje favorito, Python. Sharlynn Cobaugh y Paula Steuer fueron ayudantes decisivos, evitando que Bruce fuese sumergido en una avalancha de proyectos.

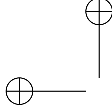
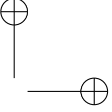
La pareja de Bruce Dawn McGee aportó una inspiración muy valiosa y un gran entusiasmo durante este proyecto. El elenco de amigos que han ayudado, pero no limitado a ellos: Mark Western, Gen Kiyooka, Kraig Brockschmidt, Zack Urlocker, Andrew Binstock, Neil Rubenking, Steve Sinofsky, JD Hildebrandt, Brian McElhinney, Brinkley Barr, Bill Gates en el Midnight Engineering Magazine, Larry Constantine y Lucy Lockwood, Tom Keffer, Greg Perry, Dan Putterman, Christi Westphal, Gene Wang, Dave Mayer, David Intersimone, Claire Sawyers, los italianos (Andrea Provaglio, Laura Fallai, Marco Cantu, Corrado, Ilsa and Christina Giustozzi), Chris y Laura Strand, The Almquists, Brad Jerbic, John Kruth y Marilyn Cvitanic, Holly Payne (¡sí, el famoso novelista!), Mark Mabry, The Robbins Families, The Moelter Families (y the McMillans), The Wilks, Dave Stoner, Laurie Adams, The Cranstons, Larry Fogg, Mike y Karen Sequeira, Gary Entsminger y Allison Brody, Chester Andersen, Joe Lordi, Dave y Brenda Bartlett, The Rentschlers, The Sudeks, Lynn y Todd, y sus familias. Y por supuesto, mamá y papá, Sandy, James y Natalie, Kim y Jared, Isaac, y Abbi.

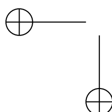
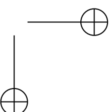



Parte I

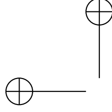
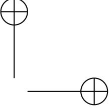
Construcción de Sistemas estables

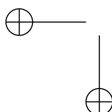
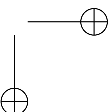



Los ingenieros de software gastan tanto tiempo en validar código como el que tardan en crearlo. La calidad es, o debería ser, el objetivo de todo programador, y se puede recorrer un largo camino hacia ese objetivo eliminando problemas antes de que aparezcan. Además, los sistemas software deberían ser lo suficientemente robustos como para comportarse razonablemente en presencia de problemas imprevistos.

Las excepciones se introdujeron en C++ para facilitar una gestión de errores sofisticada sin trocear el código con una innumerable cantidad de lógica de error. El Capítulo 1 explica cómo el uso apropiado de las excepciones puede hacer software FIXME:well-behaved, y también introduce los principios de diseño que subyacen al código seguro. En el Capítulo 2 cubrimos las pruebas unitarias y las técnicas de depuración que prevén maximizar la calidad del código antes de ser entregado. El uso de aserciones para expresar y reforzar las invariantes de un programa es una señal inequívoca de un ingeniero de software experimentado. También introducimos un entorno simple para dar soporte a las pruebas unitarias.

2: Tratamiento de excepciones

Mejorar la recuperación de errores es una de las maneras más potentes de incrementar la robustez de su código.

Una de las principales características de C++ es el tratamiento o manejo de excepciones, el cual es una manera mejor de pensar acerca de los errores y su tratamiento. Con el tratamiento de excepciones:

1. El código de manejo de errores no resulta tan tedioso de escribir y no se entremezcla con su código «normal». Usted escribe el código que desea que se ejecute, y más tarde, en una sección aparte, el código que se encarga de los problemas. Si realiza varias llamadas a la misma función, el manejo de errores de esa función se hará una sola vez, en un solo lugar.

2. Los errores no pueden ser ignorados. Si una función necesita enviar un mensaje de error al invocador de esa función, ésta «lanza» un objeto que representa a ese error fuera de la función. Si el invocador no «captura» el error y lo trata, éste pasa al siguiente ámbito abarcador, y así hasta que el error es capturado o el programa termina al no existir un manejador adecuado para ese tipo de excepción.

2.1. Tratamiento tradicional de errores

En la mayoría de ejemplos de estos volúmenes, usamos la función assert() para lo que fue concebida: para la depuración durante el desarrollo insertando código que puede deshabilitarse con #define NDEBUG en un producto comercial. Para la comprobación de errores en tiempo de ejecución se utilizan las funciones de require.h (assure( ) y require( )) desarrolladas en el capítulo 9 del Volumen 1 y repetidas aquí en el Apéndice B. Estas funciones son un modo conveniente de decir, «Hay un problema aquí que probablemente quiera manejar con un código algo más sofisticado, pero no es necesario que se distraiga con eso en este ejemplo.» Las funciones de require.h pueden parecer suficientes para programas pequeños, pero para productos complicados deseará escribir un código de manejo de errores más sofisticado.

El tratamiento de errores es bastante sencillo cuando uno sabe exactamente qué hacer, puesto que se tiene toda la información necesaria en ese contexto. Simplemente se trata el error en ese punto.

El problema ocurre cuando no se tiene suficiente información en ese contexto, y se necesita pasar la información sobre el error a un contexto diferente donde esa información sí que existe. En C, esta situación puede tratarse usando tres enfoques:

2. Usar el poco conocido sistema de manejo de señales de la biblioteca estándar de C, implementado en las funciones signal( ) (para determinar lo que ocurre cuando se presenta un evento) y raise( ) (para generar un evento). De nuevo, esta alternativa supone un alto acoplamiento debido a que requiere que el usuario de cualquier

biblioteca que genere señales entienda e instale el mecanismo de manejo de señales adecuado. En proyectos grandes los números de las señales de las diferentes bibliotecas puede llegar a entrar en conflicto.

Cuando se consideran los esquemas de tratamiento de errores para C++, hay un problema adicional que es crítico: Las técnicas de C de señales y setjmp( )/longjmp( ) no llaman a los destructores, por lo que los objetos no se limpian adecuadamente. (De hecho, si longjmp( ) salta más allá del final de un ámbito donde los destructores deben ser llamados, el comportamiento del programa es indefinido.) Esto hace casi imposible recuperarse efectivamente de una condición excepcional, puesto que siempre se están dejando objetos detrás sin limpiar y a los que ya no se tiene acceso. El siguiente ejemplo lo demuestra con setjmp/longjmp:

```
//: C01:Nonlocal.cpp
// setjmp() & longjmp().
#include <iostream>
#include <csetjmp>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

jmp_buf kansas;

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home" << endl;
  longjmp(kansas, 47);
}

int main() {
  if(setjmp(kansas) == 0) {
    cout << "tornado, witch, munchkins..." << endl;
    oz();
  } else {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
         << endl;
  }
} ///:~
```

El problema con C++ es que longjmp( ) no respeta los objetos; en particular no llama a los destructores cuando salta fuera de un ámbito.[1] Puesto que las llamadas a los destructores son esenciales, esta propuesta no es válida para C++. De hecho, el estándar de C++ aclara que saltar a un ámbito con goto (pasando por alto las llamadas a los constructores), o saltar fuera de un ámbito con longjmp( ) donde un objeto en la pila posee un destructor, constituye un comportamiento indefinido.

2.2. Lanzar una excepción

Si usted se encuentra en su código con una situación excepcional-es decir, si no tiene suficiente información en el contexto actual para decidir lo que hacer- puede enviar información acerca del error a un contexto mayor creando un objeto que contenga esa información y «lanzándolo» fuera de su contexto actual. Esto es lo que se llama lanzar una excepción. Este es el aspecto que tiene:

```
//: C01:MyError.cpp {RunByHand}

class MyError {
  const char* const data;
public:
  MyError(const char* const msg = 0) : data(msg) {}
};

void f() {
  // Here we "throw" an exception object:
  throw MyError("something bad happened");
}

int main() {
  // As you'll see shortly, we'll want a "try block" here:
  f();
} ///:~
```

MyError es una clase normal, que en este caso acepta un char* como argumento del constructor. Usted puede usar cualquier tipo para lanzar (incluyendo los tipos predefinidos), pero normalmente creará clases especial para lanzar excepciones.

La palabra clave throw hace que suceda una serie de cosas relativamente mágicas. En primer lugar se crea una copia del objeto que se está lanzando y se «devuelve» desde la función que contiene la expresión throw, aun cuando ese tipo de objeto no es lo que normalmente la función está diseñada para devolver. Un modo simplificado de pensar acerca del tratamiento de excepciones es como un mecanismo alternativo de retorno (aunque llegará a tener problemas si lleva esta analogía demasiado lejos). También es posible salir de ámbitos normales lanzando una excepción. En cualquier caso se devuelve un valor y se sale de la función o ámbito.

Además es posible lanzar tantos tipos de objetos diferentes como se quiera. Típicamente, para cada categoría de error se lanzará un tipo diferente. La idea es almacenar la información en el objeto y en el nombre de la clase con el fin de quien esté en el contexto invocador pueda averiguar lo que hacer con esa excepción.

2.3. Capturar una excepción

2.3.1. El bloque `try`

```
try {
    // Code that may generate exceptions
}
```

2.3.2. Manejadores de excepción

```
try {
    // Code that may generate exceptions
} catch(type1 id1) {
    // Handle exceptions of type1
} catch(type2 id2) {
    // Handle exceptions of type2
} catch(type3 id3)
    // Etc...
} catch(typeN idN)
    // Handle exceptions of typeN
}
// Normal execution resumes here...
```

```
//: C01:Nonlocal2.cpp
// Illustrates exceptions.
#include <iostream>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home" << endl;
  throw 47;
}

int main() {
  try {
    cout << "tornado, witch, munchkins..." << endl;
    oz();
  } catch(int) {
    cout << "Auntie Em! I had the strangest dream..."
         << endl;
  }
} ///:~
```

2.3.3.

2.4.

```
//: C01:Autoexcp.cpp
// No matching conversions.
#include <iostream>
using namespace std;
```

```
class Except1 {};

class Except2 {
public:
  Except2(const Except1&) {}
};

void f() { throw Except1(); }

int main() {
  try { f();
  } catch(Except2&) {
    cout << "inside catch(Except2)" << endl;
  } catch(Except1&) {
    cout << "inside catch(Except1)" << endl;
  }
} ///:~
```

```
//: C01:Basexcpt.cpp
// Exception hierarchies.
#include <iostream>
using namespace std;

class X {
public:
  class Trouble {};
  class Small : public Trouble {};
  class Big : public Trouble {};
  void f() { throw Big(); }
};

int main() {
  X x;
  try {
    x.f();
  } catch(X::Trouble&) {
    cout << "caught Trouble" << endl;
  // Hidden by previous handler:
  } catch(X::Small&) {
    cout << "caught Small Trouble" << endl;
  } catch(X::Big&) {
    cout << "caught Big Trouble" << endl;
  }
} ///:~
```

2.4.1. Capturar cualquier excepción

```
catch(...) {
    cout << "an exception was thrown" << endl;
}
```

2.4.2. Relanzar una excepción

```
catch(...) {
cout << "an exception was thrown" << endl;
// Deallocate your resource here, and then rethrow
    throw;
}
```

2.4.3. Excepciones no capturadas

```
//: C01:Terminator.cpp
// Use of set_terminate(). Also shows uncaught exceptions.
#include <exception>
#include <iostream>
using namespace std;

void terminator() {
  cout << "I'll be back!" << endl;
  exit(0);
}

void (*old_terminate)() = set_terminate(terminator);

class Botch {
public:
  class Fruit {};
  void f() {
    cout << "Botch::f()" << endl;
    throw Fruit();
  }
  ~Botch() { throw 'c'; }
};

int main() {
  try {
    Botch b;
    b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
} ///:~
```

2.5. Limpieza

```
//: C01:Cleanup.cpp
// Exceptions clean up complete objects only.
#include <iostream>
using namespace std;

class Trace {
  static int counter;
```

```
  int objid;
public:
  Trace() {
    objid = counter++;
    cout << "constructing Trace #" << objid << endl;
    if(objid == 3) throw 3;
  }
  ~Trace() {
    cout << "destructing Trace #" << objid << endl;
  }
};

int Trace::counter = 0;

int main() {
  try {
    Trace n1;
    // Throws exception:
    Trace array[5];
    Trace n2;  // Won't get here.
  } catch(int i) {
    cout << "caught " << i << endl;
  }
} ///:~
```

```
constructing Trace #0
constructing Trace #1
constructing Trace #2
constructing Trace #3
destructing Trace #2
destructing Trace #1
destructing Trace #0
caught 3
```

2.5.1. Gestión de recursos

```
//: C01:Rawp.cpp
// Naked pointers.
#include <iostream>
#include <cstddef>
using namespace std;

class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
  void* operator new(size_t sz) {
    cout << "allocating a Dog" << endl;
    throw 47;
  }
  void operator delete(void* p) {
    cout << "deallocating a Dog" << endl;
```

```
    ::operator delete(p);
  }
};

class UseResources {
  Cat* bp;
  Dog* op;
public:
  UseResources(int count = 1) {
    cout << "UseResources()" << endl;
    bp = new Cat[count];
    op = new Dog;
  }
  ~UseResources() {
    cout << "~UseResources()" << endl;
    delete [] bp; // Array delete
    delete op;
  }
};

int main() {
  try {
    UseResources ur(3);
  } catch(int) {
    cout << "inside handler" << endl;
  }
} ///:~
```

```
UseResources()
Cat()
Cat()
Cat()
allocating a Dog
inside handler
```

2.5.2.

```
//: C01:Wrapped.cpp
// Safe, atomic pointers.
#include <iostream>
#include <cstddef>
using namespace std;

// Simplified. Yours may have other arguments.
template<class T, int sz = 1> class PWrap {
  T* ptr;
public:
  class RangeError {}; // Exception class
  PWrap() {
    ptr = new T[sz];
    cout << "PWrap constructor" << endl;
  }
  ~PWrap() {
    delete[] ptr;
    cout << "PWrap destructor" << endl;
```

```
  }
  T& operator[](int i) throw(RangeError) {
    if(i >= 0 && i < sz) return ptr[i];
    throw RangeError();
  }
};

class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
  void g() {}
};

class Dog {
public:
  void* operator new[](size_t) {
    cout << "Allocating a Dog" << endl;
    throw 47;
  }
  void operator delete[](void* p) {
    cout << "Deallocating a Dog" << endl;
    ::operator delete[](p);
  }
};

class UseResources {
  PWrap<Cat, 3> cats;
  PWrap<Dog> dog;
public:
  UseResources() { cout << "UseResources()" << endl; }
  ~UseResources() { cout << "~UseResources()" << endl; }
  void f() { cats[1].g(); }
};

int main() {
  try {
    UseResources ur;
  } catch(int) {
    cout << "inside handler" << endl;
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
} ///:~
```

```
Cat()
Cat()
Cat()
PWrap constructor
allocating a Dog
~Cat()
~Cat()
~Cat()
PWrap destructor
inside handler
```

2.5.3. auto_ptr

```
//: C01:Auto_ptr.cpp
// Illustrates the RAII nature of auto_ptr.
#include <memory>
#include <iostream>
#include <cstddef>
using namespace std;

class TraceHeap {
  int i;
public:
  static void* operator new(size_t siz) {
    void* p = ::operator new(siz);
    cout << "Allocating TraceHeap object on the heap "
         << "at address " << p << endl;
    return p;
  }
  static void operator delete(void* p) {
    cout << "Deleting TraceHeap object at address "
         << p << endl;
    ::operator delete(p);
  }
  TraceHeap(int i) : i(i) {}
  int getVal() const { return i; }
};

int main() {
  auto_ptr<TraceHeap> pMyObject(new TraceHeap(5));
  cout << pMyObject->getVal() << endl;  // Prints 5
} ///:~
```

2.5.4. Bloques try a nivel de función

```
//: C01:InitExcept.cpp {-bor}
// Handles exceptions from subobjects.
#include <iostream>
using namespace std;

class Base {
  int i;
public:
  class BaseExcept {};
  Base(int i) : i(i) { throw BaseExcept(); }
};

class Derived : public Base {
public:
  class DerivedExcept {
    const char* msg;
  public:
    DerivedExcept(const char* msg) : msg(msg) {}
    const char* what() const { return msg; }
  };
```

```
  Derived(int j) try : Base(j) {
    // Constructor body
    cout << "This won't print" << endl;
  } catch(BaseExcept&) {
    throw DerivedExcept("Base subobject threw");;
  }
};

int main() {
  try {
    Derived d(3);
  } catch(Derived::DerivedExcept& d) {
    cout << d.what() << endl;  // "Base subobject threw"
  }
} ///:~
```

```
//: C01:FunctionTryBlock.cpp {-bor}
// Function-level try blocks.
// {RunByHand} (Don't run automatically by the makefile)
#include <iostream>
using namespace std;

int main() try {
  throw "main";
} catch(const char* msg) {
   cout << msg << endl;
   return 1;
} ///:~
```

2.6. Excepciones estándar

```
//: C01:StdExcept.cpp
// Derives an exception class from std::runtime_error.
#include <stdexcept>
#include <iostream>
using namespace std;

class MyError : public runtime_error {
public:
  MyError(const string& msg = "") : runtime_error(msg) {}
};

int main() {
  try {
    throw MyError("my message");
  } catch(MyError& x) {
    cout << x.what() << endl;
  }
} ///:~
```

2.7. Especificaciones de excepciones

```
void f() throw(toobig, toosmall, divzero);
```

```
void f();
```

```
void f() throw();
```

```
//: C01:Unexpected.cpp
// Exception specifications & unexpected(),
//{-msc} (Doesn't terminate properly)
#include <exception>
#include <iostream>
using namespace std;

class Up {};
class Fit {};
void g();

void f(int i) throw(Up, Fit) {
  switch(i) {
    case 1: throw Up();
    case 2: throw Fit();
  }
  g();
}

// void g() {}         // Version 1
void g() { throw 47; } // Version 2

void my_unexpected() {
  cout << "unexpected exception thrown" << endl;
  exit(0);
}

int main() {
  set_unexpected(my_unexpected); // (Ignores return value)
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    } catch(Up) {
      cout << "Up caught" << endl;
    } catch(Fit) {
      cout << "Fit caught" << endl;
    }
} ///:~
```

```
//: C01:BadException.cpp {-bor}
#include <exception>    // For std::bad_exception
#include <iostream>
#include <cstdio>
```

```
using namespace std;

// Exception classes:
class A {};
class B {};

// terminate() handler
void my_thandler() {
  cout << "terminate called" << endl;
  exit(0);
}

// unexpected() handlers
void my_uhandler1() { throw A(); }
void my_uhandler2() { throw; }

// If we embed this throw statement in f or g,
// the compiler detects the violation and reports
// an error, so we put it in its own function.
void t() { throw B(); }

void f() throw(A) { t(); }
void g() throw(A, bad_exception) { t(); }

int main() {
  set_terminate(my_thandler);
  set_unexpected(my_uhandler1);
  try {
    f();
  } catch(A&) {
    cout << "caught an A from f" << endl;
  }
  set_unexpected(my_uhandler2);
  try {
    g();
  } catch(bad_exception&) {
    cout << "caught a bad_exception from g" << endl;
  }
  try {
    f();
  } catch(...) {
    cout << "This will never print" << endl;
  }
} ///:~
```

2.7.1. ¿Mejores especificaciones de excepciones?

```
void f();
```

```
 void f() throw(...); // Not in C++
```

2.7.2. Especificación de excepciones y herencia

```
//: C01:Covariance.cpp {-xo}
// Should cause compile error. {-mwcc}{-msc}
#include <iostream>
using namespace std;

class Base {
public:
  class BaseException {};
  class DerivedException : public BaseException {};
  virtual void f() throw(DerivedException) {
    throw DerivedException();
  }
  virtual void g() throw(BaseException) {
    throw BaseException();
  }
};

class Derived : public Base {
public:
  void f() throw(BaseException) {
    throw BaseException();
  }
  virtual void g() throw(DerivedException) {
    throw DerivedException();
  }
}; ///:~
```

2.7.3. Cuándo no usar especificaciones de excepción

```
T pop() throw(logic_error);
```

2.8. Seguridad de la excepción

```
void pop();
```

```
template<class T> T stack<T>::pop() {
  if(count == 0)
    throw logic_error("stack underflow");
  else
    return data[--count];
}
```

```
//: C01:SafeAssign.cpp
// An Exception-safe operator=.
#include <iostream>
#include <new>       // For std::bad_alloc
```

```
#include <cstring>
#include <cstddef>
using namespace std;

// A class that has two pointer members using the heap
class HasPointers {
  // A Handle class to hold the data
  struct MyData {
    const char* theString;
    const int* theInts;
    size_t numInts;
    MyData(const char* pString, const int* pInts,
      size_t nInts)
    : theString(pString), theInts(pInts), numInts(nInts) {}
  } *theData;  // The handle
  // Clone and cleanup functions:
  static MyData* clone(const char* otherString,
      const int* otherInts, size_t nInts) {
    char* newChars = new char[strlen(otherString)+1];
    int* newInts;
    try {
      newInts = new int[nInts];
    } catch(bad_alloc&) {
      delete [] newChars;
      throw;
    }
    try {
      // This example uses built-in types, so it won't
      // throw, but for class types it could throw, so we
      // use a try block for illustration. (This is the
      // point of the example!)
      strcpy(newChars, otherString);
      for(size_t i = 0; i < nInts; ++i)
        newInts[i] = otherInts[i];
    } catch(...) {
      delete [] newInts;
      delete [] newChars;
      throw;
    }
    return new MyData(newChars, newInts, nInts);
  }
  static MyData* clone(const MyData* otherData) {
    return clone(otherData->theString, otherData->theInts,
                 otherData->numInts);
  }
  static void cleanup(const MyData* theData) {
    delete [] theData->theString;
    delete [] theData->theInts;
    delete theData;
  }
public:
  HasPointers(const char* someString, const int* someInts,
              size_t numInts) {
    theData = clone(someString, someInts, numInts);
  }
  HasPointers(const HasPointers& source) {
    theData = clone(source.theData);
  }
```

```
  HasPointers& operator=(const HasPointers& rhs) {
    if(this != &rhs) {
      MyData* newData = clone(rhs.theData->theString,
        rhs.theData->theInts, rhs.theData->numInts);
      cleanup(theData);
      theData = newData;
    }
    return *this;
  }
  ~HasPointers() { cleanup(theData); }
  friend ostream&
  operator<<(ostream& os, const HasPointers& obj) {
    os << obj.theData->theString << ": ";
    for(size_t i = 0; i < obj.theData->numInts; ++i)
      os << obj.theData->theInts[i] << ' ';
    return os;
  }
};

int main() {
  int someNums[] = { 1, 2, 3, 4 };
  size_t someCount = sizeof someNums / sizeof someNums[0];
  int someMoreNums[] = { 5, 6, 7 };
  size_t someMoreCount =
  sizeof someMoreNums / sizeof someMoreNums[0];
  HasPointers h1("Hello", someNums, someCount);
  HasPointers h2("Goodbye", someMoreNums, someMoreCount);
  cout << h1 << endl;  // Hello: 1 2 3 4
  h1 = h2;
  cout << h1 << endl;  // Goodbye: 5 6 7
} ///:~
```

2.9. Programar con excepciones

2.9.1. Cuándo evitar las excepciones

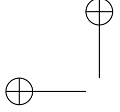
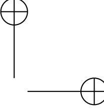
2.9.2. Usos típicos de excepciones

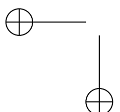
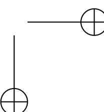
2.10. Sobrecarga

```
//: C01:HasDestructor.cpp {O}
class HasDestructor {
public:
  ~HasDestructor() {}
};

void g(); // For all we know, g may throw.

void f() {
  HasDestructor h;
  g();
} ///:~
```

2.11. Resumen

2.12. Ejercicios

3: Programación defensiva

Escribir software puede ser un objetivo difícil para desarrolladores, pero unas pocas técnicas defensivas, aplicadas rutinariamente, pueden dirigir a un largo camino hacia la mejora de la calidad de su código.

Aunque la complejidad de la producción típica de software garantiza que los probadores tendrán siempre trabajo, esperamos que anheles producir software sin defectos. Las técnicas de diseño orientada a objetos hacen mucho para limitar la dificultad de proyectos grandes, pero finalmente debe escribir bucles y funciones. Estos pequeños detalles de programación se convierten en los bloques de construcción de componentes mayores necesarios para sus diseños. Si sus blucles fallan por uno o sus funciones calculan los valores correctos sólo la mayoría de las veces, tiene problemas no importa como de elaborada sea su metodología general. En este capítulo, verá prácticas que ayudan a crear código robusto sin importar el tamaño de su proyecto.

Su código es, entre otras cosas, una expresión de su intento de resolver un problema. Sería claro para el lector (incluyendo usted) exactamente lo que estaba pensando cuando diseño aquel bucle. En ciertos puntos de su programa, deberá crear atreverse con sentencias que considera alguna u otra condición. (Si no puede, no ha realmente solucionado todavía el problema.) Tales sentencias se llaman invariantes, puesto que deberían ser invariablemente verdad en el punto donde aparecen en el código; si no, o su diseño es defectuoso, o su código no refleja con precisión su diseño.

Considere un programa que juega al juego de adivinanza mayor-menor. Una persona piensa un número entre el 1 y 100,y la otra persona adivina el número. (Permitirá al ordenador hacer la adivinanza.) La persona que piensa el número le dice al adivinador si su conjetura es mayor, menor o correcta. La mejor estrategia para el adivinador es la búsqueda binaria, que elige el punto medio del rango de los números donde el número buscado reside. La respuesta mayor-menor dice al adivinador que mitad de la lista ocupa el número, y el proceso se repite, reduciendo el tamaño del rango de búsqueda activo en cada iteración. ¿Entonces cómo escribe un bucle para realizar la repetición correctamente? No es suficiente simplemente decir

bool adivinado = false;

while(!adivinado) { ... }

porque un usuario malintencionado podría responder engañosamente, y podría pasarse todo el día adivinando. ¿ Qué suposición, que sea sencilla, está haciendo cada vez que adivina? En otras palabras, ¿qué condición debería cumplir por diseño en cada iteración del bucle?

La suposición sencilla es que el número secreto está dentro del actual rango activo de números sin adivinar: [1, 100]. Suponga que etiquetamos los puntos finales del rango con las variables bajo y alto. Cada vez que pasa por el bucle necesita asegurarse que si el número estaba en el rango [bajo, alto] al principio del bucle, calcule

el nuevo rango de modo que todavía contenga el número al final de la iteración en curso.

El objetivo es expresar el invariante del bucle en código de modo que una violación pueda ser detectada en tiempo de ejecución. Desafortunadamente, ya que el ordenador no conoce el número secreto, no puede expresar esta condición directamente en código, pero puede al menos hacer un comentario para este efecto:

while(!adivinado) { // INVARIANTE: el número está en el rango [low, high]

¿Qué ocurre cuando el usuario dice que una conjetura es demasiado alta o demasiado baja cuando no lo es? El engaño excluiría el número secreto del nuevo subrango. Porque una mentira siempre dirige a otra, finalmente su rango disminuirá a nada (puesto que se reduce a la mitad cada vez y el número secreto no está allí). Podemos expresar esta condición en el siguiente programa:

```
//: C02:HiLo.cpp {RunByHand}
// Plays the game of Hi-Lo to illustrate a loop invariant.
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

int main() {
  cout << "Think of a number between 1 and 100" << endl
        << "I will make a guess; "
        << "tell me if I'm (H)igh or (L)ow" << endl;
  int low = 1, high = 100;
  bool guessed = false;
  while(!guessed) {
    // Invariant: the number is in the range [low, high]
    if(low > high) {  // Invariant violation
      cout << "You cheated! I quit" << endl;
      return EXIT_FAILURE;
    }
    int guess = (low + high) / 2;
    cout << "My guess is " << guess << ". ";
    cout << "(H)igh, (L)ow, or (E)qual? ";
    string response;
    cin >> response;
    switch(toupper(response[0])) {
      case 'H':
        high = guess - 1;
        break;
      case 'L':
        low = guess + 1;
        break;
      case 'E':
        guessed = true;
        break;
      default:
        cout << "Invalid response" << endl;
        continue;
    }
  }
  cout << "I got it!" << endl;
  return EXIT_SUCCESS;
} ///:~
```

La violación del invariante se detecta con la condición if(menor > mayor), porque si el usuario siempre dice la verdad, siempre encontraremos el número secreto antes que agotásemos los intentos.

Usamos también una técnica del estándar C para informar sobre el estado de un programa al contexto llamante devolviendo diferentes valores desde main( ). Es portable para usar la sentencia return 0; para indicar éxito, pero no hay un valor portable para indicar fracaso. Por esta razón usamos la macro declarada para este propósito en <cstdlib>:EXIT_FAILURE. Por consistencia, cuando usamos EXIT_FAILURE también usamos EXIT_SUCCESS, a pesar de que éste es siempre definido como cero.

3.1. Aserciones

La condición en el programa mayor-menor depende de la entrada del usuario, por lo tanto no puede prevenir una violación del invariante. Sin embargo, los invariantes normalmente dependen solo del código que escribe, por eso comprobarán siempre si ha implementado su diseño correctamente. En este caso, es más claro hacer una aserción, que es un sentencia positiva que muestra sus decisiones de diseño.

Suponga que está implementando un vector de enteros: un array expandible que crece a petición. La función que añade un elemento al vector debe primero verificar que hay un espacio vacío en el array subyacente que contiene los elementos; de lo contrario, necesita solicitar más espacio en la pila y copiar los elementos existentes al nuevo espacio antes de añadir el nuevo elemento (y borrar el viejo array). Tal función podría ser de la siguiente forma:

```
void MyVector::push_back(int x) {
if(nextSlot == capacity)
grow();
assert(nextSlot < capacity);
data[nextSlot++] = x;
}
```

En este ejemplo, la información es un array dinámico de ints con capacidad espacios y espacioSiguiente espacios en uso. El propósito de grow( ) es expandir el tamaño de la información para que el nuevo valor de capacidad sea estrictamente mayor que espacioSiguiente. El comportamiento correcto de MiVector depende de esta decisión de diseño, y nunca fallará si el resto del código secundario es correcto. Afirmamos la condición con la macro assert( ), que está definido en la cabecera <cassert>.

La macro assert( ) de la biblioteca Estándar de C es breve, que resulta, portable. Si la condición en su parámetro no evalúa a cero, la ejecución continúa ininterrumpidamente; si no, un mensaje contiene el texto de la expresión culpable con su nombre de fichero fuente y el número de línea impreso en el canal de error estándar y el programa se suspende. ¿Es eso tan drástico? En la práctica, es mucho más drástico permitir que la ejecución continue cuando un supuesto de diseño básico ha fracasado. Su programa necesita ser arreglado.

Si todo va bien, probará a conciencia su código con todas las aserciones intactas hasta el momento en que se haga uso del producto final. (Diremos más sobre pruebas más tarde.) Depende de la naturaleza de su aplicación, los ciclos de máquina necesarios para probar todas las aserciones en tiempo de ejecución podrían tener

demasiado impacto en el rendimiento en producción. En ese caso, puede eliminar todas las aserciones del código automáticamente definiendo la macro NDEBUG y reconstruir la aplicación.

Para ver como funciona esto, observe que una implementación típica de assert( ) se parece a esto:

```
#ifdef NDEBUG
#define assert(cond) ((void)0)
#else
void assertImpl(const char*, const char*, long);
#define assert(cond) \
((cond) ? (void)0 : assertImpl(???))
#endif
```

Cuando la macro NDEBUG está definida, el código se descompone a la expresión (void) 0, todo lo que queda en la cadena de compilación es una sentencia esencialmente vacía como un resultado de la semicolumna que añade a cada invocación de assert( ). Si NDEBUG no está definido, assert(cond) se expande a una sentencia condicional que, cuando cond es cero, llama a una función dependiente del compilador (que llamamos assertImpl( )) con argumento string representando el texto de cond, junto con el nombre de fichero y el número de línea donde aparece la aserción. (Usamos como un marcador de posición en el ejemplo, pero la cadena mencionada es de hecho computada allí, junto con el nombre del fichero y el número de línea donde la macro aparece en ese fichero. Como estos valores se obtienen es irrelevante para nuestra discusión.) Si quiere activar y desactivar aserciones en diferentes puntos de su programa, no debe solo #define o #undef NDEBUG, sino que debe también reincluir <cassert>. Las macros son evaluadas cuando el preprocesador los encuentra y así usa cualquier estado NDEBUG se aplica en el punto de inclusión. El camino más común define NDEBUG una vez para todo el programa es como una opción del compilador, o mediante la configuración del proyecto en su entorno visual o mediante la línea de comandos, como en:

```
mycc NDEBUG myfile.cpp
```

La mayoría de los compiladores usan la bandera para definir los nombres de las macros. (Substituya el nombre del ejecutable de su compiladores por mycc arriba.) La ventaja de este enfoque es que puede dejar sus aserciones en el código fuente como un inapreciable parte de documentación, y no hay aún castigo en tiempo de ejecución. Porque el código en una aserción desaparece cuando NDEBUG está definido, es importante que no haga trabajo en una aserción. Sólo las condiciones de prueba que no cambien el estado de su programa.

Si usar NDEBUG para liberar código es una buena idea queda un tema de debate. Tony Hoare, una de los más influyentes expertos en informática de todos los tiempos,[15] ha sugerido que desactivando las comprobaciones en tiempo de ejecución como las aserciones es similar a un entusiasta de navegación que lleva un chaleco salvavidas mientras entrena en tierra y luego se deshace de él cuando va al mar.[16] Si una aserción falla en producción, tiene un problema mucho peor que la degradación en rendimiento, así que elija sabiamente.

No todas las condiciones deberían ser cumplidas por aserciones. Los errores de usuario y los fallos de los recursos en tiempos de ejecución deberían ser señalados lanzando excepciones, como explicamos en detalle en el Capítulo 1. Es tentador usar aserciones para la mayoría de las condiciones de error mientras esbozamos código,

con el propósito de remplazar muchos de ellos después con un manejador de excepciones robusto. Como cualquier otra tentación, úsese con moderación, pues podría olvidar hacer todos los cambios necesarios más tarde. Recuerde: las aserciones tienen la intención de verificar decisiones de diseño que fallarán sólo por lógica defectuosa del programador. Lo ideal es solucionar todas las violaciones de aserciones durante el desarrollo. No use aserciones para condiciones que no están totalmente en su control (por ejemplo, condiciones que dependen de la entrada del usuario). En particular, no querría usar aserciones para validar argumentos de función; lance un logic_error en su lugar.

El uso de aserciones como una herramienta para asegurar la corrección de un programa fue formalizada por Bertran Meyer en su Diseño mediante metodología de contrato.[17] Cada función tiene un contrato implícito con los clientes que, dadas ciertas precondiciones, garantiza ciertas postcondiciones. En otras palabras, las precondiciones son los requerimientos para usar la función, como los argumentos que se facilitan dentro de ciertos rangos, y las postcondiciones son los resultados enviados por la función o por retorno por valor o por efecto colateral.

Cuando los programas clientes fallan al darle un entrada válida, debe comentarles que han roto el contrato. Este no es el mejor momento para suspender el programa (aunque está justificado hacerlo desde que el contrato fue violado), pero una excepción es desde luego apropiada. Esto es porque la librería Estándar de C++ lanza excepciones derivadas de logic_error, como out_of_range.[18] Si hay funciones que sólo usted llama, no obstante, como funciones privadas en una clase de su propio diseño, la macro assert( ) es apropiada, puesto que tiene total control sobre la situación y desde luego quiere depurar su código antes de enviarlo.

Una postcondición fallada indica un error de programa, y es apropiado usar aserciones para cualquier invariante en cualquier momento, incluyendo la postcondición de prueba al final de una función. Esto se aplica en particular a las funciones de una clase que mantienen el estado de un objeto. En el ejemplo MyVector previo, por ejemplo, un invariante razonable para todas las funciones sería:

assert(0 <= siguienteEspacio && siguienteEspacio <= capacidad);

o, si siguienteEspacio es un integer sin signo, sencillamente

assert(siguienteEspacio <= capacidad);

Tal tipo de invariante se llama invariante de clase y puede ser razonablemente forzada por una aserción. Las subclases juegan un papel de subcontratista para sus clases base porque deben mantener el contrato original entre la clase base y sus clientes. Por esta razón, las precondiciones en clases derivadas no deben imponer requerimientos adicionales más allá de aquellos del contrato base, y las postcondiciones deben cumplir al menos como mucho.[19]

Validar resultados devueltos por el cliente, sin embargo, no es más o menos que probar, de manera que usar aserciones de postcondición en este caso sería duplicar trabajo. Sí, es buena documentación, pero más de un desarrollador has sido engañado usando incorrectamente las aserciones de post-condición como un substituto para pruebas de unidad.

3.2. Un framework de pruebas unitarias sencillo

Escribir software es todo sobre encontrar requerimientos.[20] Crear estos requerimientos es difícil, y pueden cambiar de un día a otro; podría descubrir en una reunión de proyecto semanal que lo que ha empleado la semana haciendo no es

exactamente lo que los usuarios realmente quieren.

Las personas no pueden articular requerimientos de software sin muestrear un sistema de trabajo en evolución. Es mucho mejor especificar un poco, diseñar un poco, codificar un poco y probar un poco. Entonces, después de evaluar el resultado, hacerlo todo de nuevo. La habilidad para desarrollar con una moda iterativa es uno de los mejores avances del enfoque orientado a objetos, pero requiere programadores ágiles que pueden hacer código fuerte. El cambio es duro.

Otro ímpetu para el cambio viene de usted, el programador. El artífice que hay en usted quiere continuamente mejorar el diseño de su código. ¿Qué programador de mantenimiento no ha maldecido el envejecimiento, el producto de la compañía insignia como un mosaico de espaguetis inmodificable, enrevesado? La reluctancia de los supervisores en permitir que uno interfiera con un sistema que funciona le roba al código la flexibilidad que necesita para que perdure. Si no está roto, no arreglarlo finalmente le da el camino para, no podemos arreglarlo reescribámoslo. El cambio es necesario.

Afortunadamente, nuestra industria está creciendo acostumbrada a la disciplina de refactoring, el arte de reestructura internamente código para mejorar su diseño, sin cambiar su comportamiento.[21] Tales mejoras incluyen extraer una nueva función de otra, o de forma inversa, combinar funciones, reemplazar una función con un objeto; parametrizar una función o clase; y reemplazar condicionales con polimorfismo. Refactorizar ayuda al código evolucionar.

Si la fuerza para el cambio viene de los usuarios o programadores, los cambios hoy pueden destrozar lo trabajado ayer. Necesitamos un modo para construir código que resista el cambio y mejoras a lo largo del tiempo.

La Programación Extrema (XP)[22] es sólo uno de las muchas prácticas que motivan la agilidad. En esta sección exploramos lo que pensamos es la clave para hacer un desarrollo flexible, incremental que tenga éxito: un framework de pruebas unitarias automatizada fácil de usar. (Note que los probadores, profesionales de software que prueban el código de otros para ganarse la vida, son todavía indispensables. Aquí, estamos simplemente describiendo un modo para ayudar a los desarrolladores a escribir mejor código.)

Los desarrolladores escriben pruebas unitarias para conseguir confianza para decir las dos cosas más importantes que cualquier desarrollador puede decir:

1. Entiendo los requerimientos.

Mi código cumple esos requerimientos (hasta donde yo sé)

No hay mejor modo para asegurar que sabe lo que el código que está por escribir debería hacer mejor que escribir primero pruebas unitarias. Este ejercicio sencillo ayuda a centrar la mente en las tareas siguientes y probablemente guiará a código que funcionalmente más rápido mejor que sólo saltar a codificar. O, expresarlo en términos XP:

Probar + programar es más rápido que sólo programar.

Escribir primero pruebas sólo le protegen contra condiciones límite que podrían destrozar su código, por lo tanto su código es más robusto.

Cuando su código pasa todas sus pruebas, sabe que si el sistema no está funcionando, su código no es probablemente el problema. La frase todas mis pruebas funcionan es un fuerte razonamiento.

3.2.1. Pruebas automatizadas

Por lo tanto, ¿qué aspecto tiene una prueba unitaria? Demasiado a menudo los desarrolladores simplemente usan alguna entrada correcta para producir alguna salida esperada, que examinan visualmente. Existen dos peligros en este enfoque. Primero, los programas no siempre reciben sólo entradas correctas. Todos sabemos que deberíamos probar los límites de entrada de un programa, pero es duro pensar esto cuando está intentando simplemente hacer que las cosas funcionar. Si escribe primero la prueba para una función antes de comenzar a codificar, puede ponerse su traje de probador y preguntarse a si mismo, ¿qué haría posiblemente destrozar esto? Codificar una prueba que probará la función que escribirá no es erróneo, y luego ponerte el traje de desarrollador y hacerlo pasar. Escribirá mejor código que si no había escrito la prueba primero.

El segundo peligro es que esperar una salida visualmente es tedioso y propenso a error. La mayoría de cualquier tipo de cosas que un humano puede hacer un ordenador puede hacerlas, pero sin el error humano. Es mejor formular pruebas como colecciones de expresiones boolean y tener un programa de prueba que informa de cualquier fallo.

Por ejemplo, suponga que necesita construir una clase Fecha que tiene las siguientes propiedades:

Una fecha puede estar inicializada con una cadena (AAAAMMDD), 3 enteros (A, M, D), o nada (dando la fecha de hoy).

Un objecto fecha puede producir su año, mes y día o una cadena de la forma AAAAMMDD.

Todas las comparaciones relacionales están disponibles, además de calcular la duración entre dos fechas (en años, meses, y días).

Las fechas para ser comparadas necesitan poder extenderse un número arbitrario de siglos(por ejemplo, 16002200).

Su clase puede almacenar tres enteros que representan el año, mes y día. (Sólo asegúrese que el año es al menos de 16 bits de tamaño para satisfacer el último punto.) La interfaz de su clase Fecha se podría parecer a esto:

```
//: C02:Date1.h
// A first pass at Date.h.
#ifndef DATE1_H
#define DATE1_H
#include <string>

class Date {
public:
  // A struct to hold elapsed time:
  struct Duration {
    int years;
    int months;
    int days;
    Duration(int y, int m, int d)
    : years(y), months(m), days(d) {}
  };
  Date();
  Date(int year, int month, int day);
  Date(const std::string&);
  int getYear() const;
```

```
  int getMonth() const;
  int getDay() const;
  std::string toString() const;
   friend bool operator<(const Date&, const Date&);
   friend bool operator>(const Date&, const Date&);
   friend bool operator<=(const Date&, const Date&);
   friend bool operator>=(const Date&, const Date&);
   friend bool operator==(const Date&, const Date&);
   friend bool operator!=(const Date&, const Date&);
  friend Duration duration(const Date&, const Date&);
};
#endif // DATE1_H ///:~
```

Antes de que implemente esta clase, puede solidificar sus conocimientos de los requerimientos escribiendo el principio de un programa de prueba. Podría idear algo como lo siguiente:

```
//: C02:SimpleDateTest.cpp
//{L} Date
#include <iostream>
#include "Date.h" // From Appendix B
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) { if(t) nPass++; else nFail++; }

int main() {
  Date mybday(1951, 10, 1);
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
}
/* Expected output:
Passed: 3, Failed: 0
*/ ///:~
```

En este caso trivial, la función test( ) mantiene las variables globales nAprobar y nSuspender. La única revisión visual que hace es leer el resultado final. Si una prueba falla, un test( ) más sofisticado muestra un mensaje apropiado. El framework descrito más tarde en este capítulo tiene un función de prueba, entre otras cosas.

Puede ahora implementar la clase Fecha para hacer pasar estas pruebas, y luego puede proceder iterativamente hasta que se satisfagan todos los requerimientos. Escribiendo primero pruebas, es más probable que piense en casos límite que podrían destrozar su próxima implementación, y es más probable que escriba el código correctamente la primera vez. Como ejercicio podría realizar la siguiente versión de una prueba para la clase Fecha:

```
//: C02:SimpleDateTest2.cpp
//{L} Date
#include <iostream>
```

```
#include "Date.h"
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) { if(t) ++nPass; else ++nFail; }

int main() {
  Date mybday(1951, 10, 1);
  Date today;
   Date myevebday("19510930");

  // Test the operators
  test(mybday < today);
  test(mybday <= today);
  test(mybday != today);
  test(mybday == mybday);
  test(mybday >= mybday);
  test(mybday <= mybday);
  test(myevebday < mybday);
  test(mybday > myevebday);
  test(mybday >= myevebday);
  test(mybday != myevebday);

  // Test the functions
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  test(myevebday.getYear() == 1951);
  test(myevebday.getMonth() == 9);
  test(myevebday.getDay() == 30);
  test(mybday.toString() == "19511001");
  test(myevebday.toString() == "19510930");

  // Test duration
  Date d2(2003, 7, 4);
  Date::Duration dur = duration(mybday, d2);
  test(dur.years == 51);
  test(dur.months == 9);
  test(dur.days == 3);

  // Report results:
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
} ///:~
```

Esta prueba puede ser desarrollada por completo. Por ejemplo, no hemos probado que duraciones grandes son manejadas correctamente. Pararemos aquí, pero coja la idea. La implementación entera para la case Fecha está disponible en los ficheros Date.h y Date.cpp en el apéndice.[23]

3.2.2. El Framework TestSuite

Algunas herramientas de pruebas unitarias automatizadas de C++ están disponibles en la World Wide Web para descargar, como CppUnit.[24] Nuestra intención

aquí no es sólo presentar un mecanismo de prueba que sea fácil de usar, sino también fácil de entender internamente e incluso modificar si es necesario. Por lo tanto, en el espíritu de Hacer Lo Más Simple Que Podría Posiblemente Funcionar,[25] hemos desarrollado el Framework TestSuite, un espacio de nombres llamado TestSuite que contiene dos clases principales: Test y Suite.

La clase Test es una clase base abstracta de la cual deriva un objeto test. Tiene constancia del número de éxitos y fracasos y muestra el texto de cualquier condición de prueba que falla. Simplemente para sobreescribir la función run( ), que debería llamar en turnos a la macro test_() para cada condición de prueba boolean que defina.

Para definir una prueba para la clase Fecha usando el framework, puede heredar de Test como se muetra en el siguiente programa:

```
//: C02:DateTest.h
#ifndef DATETEST_H
#define DATETEST_H
#include "Date.h"
#include "../TestSuite/Test.h"

class DateTest : public TestSuite::Test {
  Date mybday;
  Date today;
  Date myevebday;
public:
  DateTest(): mybday(1951, 10, 1), myevebday("19510930") {}
  void run() {
    testOps();
    testFunctions();
    testDuration();
  }
  void testOps() {
    test_(mybday < today);
    test_(mybday <= today);
    test_(mybday != today);
    test_(mybday == mybday);
    test_(mybday >= mybday);
    test_(mybday <= mybday);
    test_(myevebday < mybday);
    test_(mybday > myevebday);
    test_(mybday >= myevebday);
    test_(mybday != myevebday);
  }
  void testFunctions() {
    test_(mybday.getYear() == 1951);
    test_(mybday.getMonth() == 10);
    test_(mybday.getDay() == 1);
    test_(myevebday.getYear() == 1951);
    test_(myevebday.getMonth() == 9);
    test_(myevebday.getDay() == 30);
    test_(mybday.toString() == "19511001");
    test_(myevebday.toString() == "19510930");
  }
  void testDuration() {
    Date d2(2003, 7, 4);
    Date::Duration dur = duration(mybday, d2);
    test_(dur.years == 51);
    test_(dur.months == 9);
```

```
    test_(dur.days == 3);
  }
};
#endif // DATETEST_H ///:~
```

Ejecutar la prueba es una sencilla cuestión de instaciación de un objeto DateTest y llamar a su función run( ):

```
//: C02:DateTest.cpp
// Automated testing (with a framework).
//{L} Date ../TestSuite/Test
#include <iostream>
#include "DateTest.h"
using namespace std;

int main() {
  DateTest test;
  test.run();
  return test.report();
}
/* Output:
Test "DateTest":
        Passed: 21,      Failed: 0
*/ ///:~
```

La función Test::report( ) muestra la salida previa y devuelve el número de fallos, de este modo es conveniente usarlo como valor de retorno desde el main( ).

La clase Test usa RTTI[26] para obtener el nombre de su clase(por ejemplo, DateTest) para el informe. Hay también una función setStream() si quiere enviar los resultados de la prueba a un fichero en lugar de la salida estándar (por defecto). Verá la implementación de la clase Test más tarde en este capítulo.

La macro test_( ) puede extraer el texto de la condición booleana que falla, junto con el nombre del fichero y número de línea.[27] Para ver lo que ocurre cuando un fallo aparece, puede insertar un error intencionado en el código, por ejemplo invirtiendo la condición en la primera llamda a test_( ) en DateTest::testOps( ) en el código de ejemplo previo. La salida indica exactamente que la prueba tenía un error y dónde ocurrió:

DateTest fallo: (mybday > hoy) , DateTest.h (línea 31) Test "DateTest": Passados: 20 Fallados: 1

Además de test_( ), el framework incluye las funciones succed_( ) y fail_( ), para casos donde una prueba Boolean no funcionará. Estas funciones se aplican cuando la clase que está probando podría lanzar excepciones. Durante la prueba, crear un conjunto de entrada que causará que la excepción aparezca. Si no, es un error y puede llamar a fail_( ) explicitamente para mostrar un mensaje y actualizar el contador de fallos. Si lanza la excecpión como se esperaba, llame a succeed_( ) para actualizar el contador de éxitos.

Para ilustrar, suponga que modificamos la especificación de los dos constructor no por defecto de Date para lanzar una excepción DateError (un tipo anidado dentro de Date y derivado de std::logic_error) si los parámetros de entrada no representa un fecha válida: Date(const string& s) throw(DateError); Date(int year, int month,

int day) throw(DateError);

La función DateTest::run( ) puede ahora llamar a la siguiente función para probar el manejo de excepciones:

```
void testExceptions() {
try {
Date d(0,0,0);  // Invalid
fail_("Invalid date undetected in Date int ctor");
} catch(Date::DateError&) {
succeed_();
}
try {
Date d("");  // Invalid
fail_("Invalid date undetected in Date string ctor");
} catch(Date::DateError&) {
succeed_();
}
}
```

En ambos casos, si una excepción no se lanza, es un error. Fíjese que debe pasar manualmente un mensaje a fail_( ), pues no se está evaluando una expresión booleana.

3.2.3. Suites de test

Los proyectos reales contienen normalmente muchas clases, por lo tanto necesita un modo para agrupar pruebas para que pueda simplemente pulsar un solo botón para probar el proyecto entero.[28] La clase Suite recoge pruebas en una unidad funcional. Añada objetos Test a Suite con la función addTest( ), o puede incluir una suite existente entera con addSuite( ). Para ilustrar, el siguiente ejemplo reúna los programas del Capítulo 3 que usa la clase Test en una sola suite. Fíjese que este fichero aparecerá en el subdirectorio del Capítulo 3:

```
//: C03:StringSuite.cpp
//{L} ../TestSuite/Test ../TestSuite/Suite
//{L} TrimTest
// Illustrates a test suite for code from Chapter 3
#include <iostream>
#include "../TestSuite/Suite.h"
#include "StringStorage.h"
#include "Sieve.h"
#include "Find.h"
#include "Rparse.h"
#include "TrimTest.h"
#include "CompStr.h"
using namespace std;
using namespace TestSuite;

int main() {
    Suite suite("String Tests");
    suite.addTest(new StringStorageTest);
    suite.addTest(new SieveTest);
    suite.addTest(new FindTest);
    suite.addTest(new RparseTest);
    suite.addTest(new TrimTest);
```

```
    suite.addTest(new CompStrTest);
    suite.run();
    long nFail = suite.report();
    suite.free();
    return nFail;
}
/* Output:
s1 = 62345
s2 = 12345
Suite "String Tests"
====================
Test "StringStorageTest":
   Passed: 2   Failed: 0
Test "SieveTest":
   Passed: 50  Failed: 0
Test "FindTest":
   Passed: 9   Failed: 0
Test "RparseTest":
   Passed: 8   Failed: 0
Test "TrimTest":
   Passed: 11  Failed: 0
Test "CompStrTest":
   Passed: 8   Failed: 0
*/ ///:~
```

5 de los tests de más arriba están completamente contenidos en los ficheros de cabecera. TrimTest no lo está, porque contiene datos estáticos que deben estar definidos en un fichero de implementación. Las dos primeras líneas de salida son trazos de la prueba StringStorage. Debe dar a la suite un nombre como argumento del constructor. La función Suite::run( ) llama a Test::run( ) po cada una de las pruebas que tiene. Más de lo mismo pasa con Suite::report( ), excepto que puede enviar los informes de pruebas individuales a cadenas destinaciones diferentes mejor que el informe de la suite. Si la prueba pasa a addSuite( ) ya tiene un puntero de cadena asignado, que lo guarda. En otro caso, obtiene su cadena del objeto Suite. (Como con Test, hay un segundo argumento opcional para el constructor suite que no se presenta a std::cout.) El destructor para Suite no borra automáticamente los punteros contenidos en Test porque no necesitan residir en la pila; este es el trabajo de Suite::free( ).

3.2.4. El código del framework de prueba

El código del framework de pruebas es un subdirectorio llamado TestSuite en la distribución de código disponible en www.MindView.net. Para usarlo, incluya la ruta de búsqueda para el subdirectorio TestSuite en la ruta de búsqueda de la biblioteca. Aquí está la cabecera para Test.h:

```
//: TestSuite:Test.h
#ifndef TEST_H
#define TEST_H
#include <string>
#include <iostream>
#include <cassert>
using std::string;
using std::ostream;
using std::cout;
```

```
// fail_() has an underscore to prevent collision with
// ios::fail(). For consistency, test_() and succeed_()
// also have underscores.

#define test_(cond) \
  do_test(cond, #cond, __FILE__, __LINE__)
#define fail_(str) \
  do_fail(str, __FILE__, __LINE__)

namespace TestSuite {

class Test {
  ostream* osptr;
  long nPass;
  long nFail;
  // Disallowed:
  Test(const Test&);
  Test& operator=(const Test&);
protected:
  void do_test(bool cond, const string& lbl,
    const char* fname, long lineno);
  void do_fail(const string& lbl,
    const char* fname, long lineno);
public:
  Test(ostream* osptr = &cout) {
    this->osptr = osptr;
    nPass = nFail = 0;
  }
  virtual ~Test() {}
  virtual void run() = 0;
  long getNumPassed() const { return nPass; }
  long getNumFailed() const { return nFail; }
  const ostream* getStream() const { return osptr; }
  void setStream(ostream* osptr) { this->osptr = osptr; }
  void succeed_() { ++nPass; }
  long report() const;
  virtual void reset() { nPass = nFail = 0; }
};

} // namespace TestSuite
#endif // TEST_H ///:~
```

Hay tres funciones virtuales en la clase Test:

Un destructor virtual

La función reset( )

La función virtual pura run( )

Como se explicó en el Volumen 1, es un error eliminar un objeto derivado de la pila a través de un puntero base a menos que la clase base tenga un destructor virtual. Cualquier clase propuesta para ser una clase base (normalmente evidenciadas por la presencia de al menos una de las otras funciones virtuales) tendría un destructor virtual. La implementación por defecto de Test::reset( ) pone los contadores de éxitos y fallos a cero. Podría querer sobreescribir esta función para poner el estado de los datos en su objeto de test derivado; sólo asegúrese de llamar a Test::rest( )

explícitamente en su sobreescritura de modo que los contadores se reajusten. La función Test::run( ) es virtual pura ya que es necesario para sobreescribirla en su clase derivada.

Las macros test_( ) y fail_( ) pueden incluir la información disponible del nombre del fichero y el número de línea del preprocesador. Originalmente omitimos el guión bajo en los nombres, pero la macro fail colisiona con ios::fail( ), provocando errores de compilación.

Aquí está la implementación del resto de las funciones Test:

```
//: TestSuite:Test.cpp {O}
#include "Test.h"
#include <iostream>
#include <typeinfo>
using namespace std;
using namespace TestSuite;

void Test::do_test(bool cond, const std::string& lbl,
  const char* fname, long lineno) {
  if(!cond)
    do_fail(lbl, fname, lineno);
  else
    succeed_();
}

void Test::do_fail(const std::string& lbl,
  const char* fname, long lineno) {
  ++nFail;
  if(osptr) {
    *osptr << typeid(*this).name()
           << "failure: (" << lbl << ") , " << fname
           << " (line " << lineno << ")" << endl;
  }
}

long Test::report() const {
  if(osptr) {
    *osptr << "Test \"" << typeid(*this).name()
           << "\":\n\tPassed: " << nPass
           << "\tFailed: " << nFail
           << endl;
  }
  return nFail;
} ///:~
```

La clase Test lleva la cuenta del número de éxitos y fracasos además de la cadena donde quiere que Test::report( ) muestre los resultados. Las macros test_( ) y fail_() extraen la información del nombre del fichero actual y el número de línea del preprocesador y pasa el nombre del fichero a do_test( ) y el número de línea a do_fail( ), que hacen el mismo trabajo de mostrar un mensaje y actualizar el contador apropiado. No podemos pensar una buena razón para permitir copiar y asignar objetos de prueba, por lo que hemos rechazado estas operaciones para hacer sus prototipos privados y omitir el cuerpo de sus respectivas funciones.

Aquí está el fichero de cabecera para Suite:

```
//: TestSuite:Suite.h
#ifndef SUITE_H
#define SUITE_H
#include <vector>
#include <stdexcept>
#include "../TestSuite/Test.h"
using std::vector;
using std::logic_error;

namespace TestSuite {

class TestSuiteError : public logic_error {
public:
  TestSuiteError(const string& s = "")
  : logic_error(s) {}
};

class Suite {
  string name;
  ostream* osptr;
  vector<Test*> tests;
  void reset();
  // Disallowed ops:
  Suite(const Suite&);
  Suite& operator=(const Suite&);
public:
  Suite(const string& name, ostream* osptr = &cout)
  : name(name) { this->osptr = osptr; }
  string getName() const { return name; }
  long getNumPassed() const;
  long getNumFailed() const;
  const ostream* getStream() const { return osptr; }
  void setStream(ostream* osptr) { this->osptr = osptr; }
  void addTest(Test* t) throw(TestSuiteError);
  void addSuite(const Suite&);
  void run();  // Calls Test::run() repeatedly
  long report() const;
  void free();  // Deletes tests
};

} // namespace TestSuite
#endif // SUITE_H ///:~
```

La clase Suite tiene punteros a sus objetos Test en un vector. Fíjese en la especificación de la excepción en la función addTest( ). Cuando añada una prueba a una suite, Suite::addTest( ) verifique que el puntero que pasa no sea null; si es null, se lanza una excepción TestSuiteError. Puesto que esto hace imposible añadir un puntero null a una suite, addSuite( ) afirma esta condición en cada prueba, como hacen las otras funciones que atraviesan el vector de pruebas (vea la siguiente implementación). Copiar y asignar están desestimados como están en la clase Test.

```
//: TestSuite:Suite.cpp {O}
#include "Suite.h"
#include <iostream>
#include <cassert>
```

```
#include <cstddef>
using namespace std;
using namespace TestSuite;

void Suite::addTest(Test* t) throw(TestSuiteError) {
  // Verify test is valid and has a stream:
  if(t == 0)
    throw TestSuiteError("Null test in Suite::addTest");
  else if(osptr && !t->getStream())
    t->setStream(osptr);
  tests.push_back(t);
  t->reset();
}

void Suite::addSuite(const Suite& s) {
   for(size_t i = 0; i < s.tests.size(); ++i) {
     assert(tests[i]);
      addTest(s.tests[i]);
  }
}

void Suite::free() {
  for(size_t i = 0; i < tests.size(); ++i) {
    delete tests[i];
    tests[i] = 0;
  }
}

void Suite::run() {
  reset();
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->run();
  }
}

long Suite::report() const {
  if(osptr) {
    long totFail = 0;
    *osptr << "Suite \"" << name
             << "\"\n=======";
    size_t i;
    for(i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=" << endl;
    for(i = 0; i < tests.size(); ++i) {
      assert(tests[i]);
      totFail += tests[i]->report();
    }
    *osptr << "=======";
    for(i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=" << endl;
    return totFail;
  }
  else
    return getNumFailed();
}
```

```
long Suite::getNumPassed() const {
  long totPass = 0;
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totPass += tests[i]->getNumPassed();
  }
  return totPass;
}

long Suite::getNumFailed() const {
  long totFail = 0;
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totFail += tests[i]->getNumFailed();
  }
  return totFail;
}

void Suite::reset() {
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->reset();
  }
} ///:~
```

Usaremos el framework TestSuite donde sea pertinente a lo largo del resto de este libro.

3.3. Técnicas de depuración

La mejor costumbre para eliminar fallos es usar aserciones como se explica al principio de este capítulo; haciendo esto le ayudará a encontrar errores lógicos antes de que causen problemas reales. Esta sección contiene otros consejos y técnicas que podrían ayudar durante la depuración.

3.3.1. Macros de seguimiento

Algunas veces es útil imprimir el código de cada sentencia cuando es ejecutada, o cout o trazar un fichero. Aquí esta una macro de preprocesaor para llevar a cabo esto:

#define TRACE(ARG) cout << #ARG << endl; ARG

Ahora puede ir a través y alrededor de las sentencias que traceé con esta macro. Sin embargo, esto puede introducir problemas. Por ejemplo, si coge la sentencia:

for(int i = 0; i < 100; i++) cout << i << endl;

y ponga ambas líneas dentro de la macro TRACE( ), obtiene esto:

TRACE(for(int i = 0; i < 100; i++)) TRACE(cout << i << endl;)

que se expande a esto:

cout << "for(int i = 0; i < 100; i++)" << endl; for(int i = 0; i < 100; i++) cout << "cout << i << endl;" << endl; cout << i << endl;

que no es exactamente lo que quiere. Por lo tanto, debe usar esta técnica cuidadosamente.

Lo siguiente es una variación en la macro TRACE( ):

#define D(a) cout << #a "=[" << a << "]" << endl;

Si quiere mostrar una expresión, simplemente póngala dentro de una llamada a D( ). La expresión se muestra, seguida de su valor (asumiendo que hay un operador sobrecargado << para el tipo de resultado). Por ejemplo, puede decir D(a + b). Puede usar esta macro en cualquier momento que quiera comprobar un valor intermedio.

Estas dos macros representan las dos cosas fundamentales que hace con un depurador: trazar la ejecución de código y mostrar valores. Un buen depurador es una herramienta de productividad excelente, pero a veces los depuradores no están disponibles, o no es conveniente usarlos. Estas técnicas siempre funcionan, sin tener en cuenta la situación.

3.3.2. Fichero de rastro

ADVERTENCIA: Esta sección y la siguiente contienen código que está oficialmente sin aprobación por el Estándar C++. En particular, redefinimos cout y new mediante macros, que puede provocar resultados sorprendentes si no tiene cuidado. Nuestros ejemplos funcionan en todos los compiladores que usamos, comoquiera, y proporcionan información útil. Este es el único lugar en este libro donde nos desviaremos de la inviolabilidad de la práctica de codificar cumpliendo el estándar. ¡Úsalo bajo tu propio riesgo! Dese cuenta que para este trabajo, usar delcaraciones debe ser realizado, para que cout no esté prefijado por su nombre de espacio, p.e. std::cout no funcionará.

El siguiente código crea fácilmente un fichero de seguimiento y envía todas las salidas que irían normalmente a cout a ese fichero. Todo lo que debe hacer es #define TRACEON e incluir el fichero de cabecera (por supuesto, es bastante fácil sólo escribir las dos líneas claves correctamente en su fichero):

```
//: C03:Trace.h
// Creating a trace file.
#ifndef TRACE_H
#define TRACE_H
#include <fstream>

#ifdef TRACEON
std::ofstream TRACEFILE__("TRACE.OUT");
#define cout TRACEFILE__
#endif

#endif // TRACE_H ///:~
```

Aquí esta una prueba sencilla del fichero anterior:

```
//: C03:Tracetst.cpp {-bor}
#include <iostream>
#include <fstream>
#include "../require.h"
using namespace std;
```

```
#define TRACEON
#include "Trace.h"

int main() {
  ifstream f("Tracetst.cpp");
  assure(f, "Tracetst.cpp");
  cout << f.rdbuf(); // Dumps file contents to file
} ///:~
```

Porque cout ha sido textualmente convertido en algo más por Trace.h, todas las sentencias cout en su programa ahora envían información al fichero de seguimiento. Esto es una forma conveniente de capturar su salida en un fichero, en caso de que su sistema operativo no haga una fácil redirección de la salida.

3.3.3. Encontrar agujeros en memoria

Las siguientes técnicas sencillas de depuración están explicadas en el Volumen 1:

1. Para comprobar los límites de un array, usa la plantilla Array en C16:Array3.cpp del Volumen 1 para todos los arrays. Puede desactivar la comprobación e incrementar la eficiencia cuando esté listo para enviar. (Aunque esto no trata con el caso de coger un puntero a un array.)

2. Comprobar destructores no virtuales en clases base. Seguirle la pista a new/-delete y malloc/free

Los problemas comunes con la asignación de memoria incluyen llamadas por error a delete para memoria que no está libre, borrar el espacio libre más de una vez, y más a menudo, olvidando borrar un puntero. Esta sección discute un sistema que puede ayudarle a localizar estos tipos de problemas.

Como cláusula adicional de exención de responsabilidad más allá de la sección precedente: por el modo que sobrecargamos new, la siguiente técnica puede no funcionar en todas las plataformas, y funcionará sólo para programas que no llaman explicitamente al operador de función new( ). Hemos sido bastante cuidadosos en este libro para presentar sólo código que se ajuste completamente al Estándar C++, pero en este ejemplo estamos haciendo una excepción por las siguientes razones:

1. A pesar de que es técnicamente ilegal, funciona en muchos compiladores.[29]

2. Ilustramos algunos pensamientos útiles en el trascurso del camino.

Para usar el sistema de comprobación de memoria, simplemente incluya el fichero de cabecera MemCheck.h, conecte el fichero MemCheck.obj a su aplicación para interceptar todas las llamadas a new y delete, y llame a la macro MEM_ON( ) (se explica más tarde en esta sección) para iniciar el seguimiento de la memoria. Un seguimiento de todas las asignaciones y desasignaciones es impreso en la salida estándar (mediante stdout). Cuando use este sistema, todas las llamadas a new almacenan información sobre el fichero y la línea donde fueron llamados. Esto está dotado usando la sintaxis de colocación para el operador new.[30] Aunque normalmente use la sintaxis de colocación cuando necesite colocar objetos en un punto de memoria específico, puede también crear un operador new( ) con cualquier número de argumentos. Esto se usa en el siguiente ejemplo para almacenar los resultados de las macros __FILE__ y __LINE__ cuando se llama a new:

```
//: C02:MemCheck.h
```

```
#ifndef MEMCHECK_H
#define MEMCHECK_H
#include <cstddef>  // For size_t

// Usurp the new operator (both scalar and array versions)
void* operator new(std::size_t, const char*, long);
void* operator new[](std::size_t, const char*, long);
#define new new (__FILE__, __LINE__)

extern bool traceFlag;
#define TRACE_ON() traceFlag = true
#define TRACE_OFF() traceFlag = false

extern bool activeFlag;
#define MEM_ON() activeFlag = true
#define MEM_OFF() activeFlag = false

#endif // MEMCHECK_H ///:~
```

Es importante incluir este fichero en cualquier fichero fuente en el que quiera seguir la actividad de la memoria libre, pero inclúyalo al final (después de sus otras directivas #include). La mayoría de las cabeceras en la biblioteca estándar son plantillas, y puesto que la mayoría de los compiladores usan el modelo de inclusión de compilación de plantilla (significa que todo el código fuente está en las cabeceras), la macro que reemplaza new en MemCheck.h usurpará todas las instancias del operador new en el código fuente de la biblioteca (y casi resultaría en errores de compilación). Además, está sólo interesado en seguir sus propios errores de memoria, no los de la biblioteca.

En el siguiente fichero, que contiene la implementación del seguimiento de memoria, todo está hecho con C estándar I/O más que con iostreams C++. No debería influir, puesto que no estamos interfiriendo con el uso de iostream en la memoria libre, pero cuando lo intentamos, algunos compiladores se quejaron. Todos los compiladores estaban felices con la versión <cstdio>.

```
//: C02:MemCheck.cpp {O}
#include <cstdio>
#include <cstdlib>
#include <cassert>
#include <cstddef>
using namespace std;
#undef new

// Global flags set by macros in MemCheck.h
bool traceFlag = true;
bool activeFlag = false;

namespace {

// Memory map entry type
struct Info {
  void* ptr;
  const char* file;
  long line;
};
```

```
// Memory map data
const size_t MAXPTRS = 10000u;
Info memMap[MAXPTRS];
size_t nptrs = 0;

// Searches the map for an address
int findPtr(void* p) {
  for(size_t i = 0; i < nptrs; ++i)
    if(memMap[i].ptr == p)
      return i;
  return -1;
}

void delPtr(void* p) {
  int pos = findPtr(p);
  assert(pos >= 0);
  // Remove pointer from map
  for(size_t i = pos; i < nptrs-1; ++i)
    memMap[i] = memMap[i+1];
  --nptrs;
}

// Dummy type for static destructor
struct Sentinel {
  ~Sentinel() {
    if(nptrs > 0) {
      printf("Leaked memory at:\n");
      for(size_t i = 0; i < nptrs; ++i)
        printf("\t%p (file: %s, line %ld)\n",
          memMap[i].ptr, memMap[i].file, memMap[i].line);
    }
    else
      printf("No user memory leaks!\n");
  }
};

// Static dummy object
Sentinel s;

} // End anonymous namespace

// Overload scalar new
void*
operator new(size_t siz, const char* file, long line) {
  void* p = malloc(siz);
  if(activeFlag) {
    if(nptrs == MAXPTRS) {
      printf("memory map too small (increase MAXPTRS)\n");
      exit(1);
    }
    memMap[nptrs].ptr = p;
    memMap[nptrs].file = file;
    memMap[nptrs].line = line;
    ++nptrs;
  }
  if(traceFlag) {
    printf("Allocated %u bytes at address %p ", siz, p);
    printf("(file: %s, line: %ld)\n", file, line);
```

```
  }
  return p;
}

// Overload array new
void*
operator new[](size_t siz, const char* file, long line) {
  return operator new(siz, file, line);
}

// Override scalar delete
void operator delete(void* p) {
  if(findPtr(p) >= 0) {
    free(p);
    assert(nptrs > 0);
    delPtr(p);
    if(traceFlag)
      printf("Deleted memory at address %p\n", p);
  }
  else if(!p && activeFlag)
    printf("Attempt to delete unknown pointer: %p\n", p);
}

// Override array delete
void operator delete[](void* p) {
  operator delete(p);
} ///:~
```

Las banderas booleanas de traceFalg y activeFlag son globales, por lo que pueden ser modificados en su código por las macros TRACE_ON( ), TRACE_OFF( ), MEM_ON( ), y MEM_OFF( ). En general, encierre todo el código en su main( ) dentro una pareja MEM_ON( )-MEM_OFF( ) de modo que la memoria sea siempre trazada. Trazar, que repite la actividad de las funciones de sustitución por el operador new( ) y el operador delete( ), es por defecto, pero puede desactivarlo con TRACE_OFF( ). En cualquier caso, los resultados finales son siempre impresos (vea la prueba que se ejecuta más tarde en este capítulo).

La facilidad MemCheck rastrea la memoria guardando todas las direcciones asignadas por el operador new( ) en un array de estructuras Info, que también tiene el nombre del fichero y el número de línea donde la llamada new se encuentra. Para prevenir la colisión con cualquier nombre que haya colocado en el espacio de nombres global, tanta información como sea posible se guarda dentro del espacio de nombre anónimo. La clase Sentinel existe únicamente para llamar a un destructor de objetos con estático cuando el programa termina. Este destructor inspecciona memMap para ver si algún puntero está esperando a ser borrado (indicando una perdida de memoria).

Nuestro operador new( ) usa malloc( ) para conseguir memoria, y luego añade el puntero y su información de fichero asociado a memMap. La función de operador delete( ) deshace todo el trabajo llamando a free( ) y decrementando nptrs, pero primero se comprueba para ver si el puntero en cuestión está en el mapa en el primer lugar. Si no es así, o reintenta borrar una dirección que no está en el almacén libre, o re intenta borrar la que ya ha sido borrada y eliminada del mapa. La variable activeFlag es importante aquí porque no queremos procesar ninguna desasignación de alguna actividad del cierre del sistema. Llamando a MEM_OFF( ) al final de su código, activeFlag será puesta a falso, y posteriores llamadas para borrar serán ig-

noradas. (Está mal en un programa real, pero nuestra intención aquí es encontrar agujeros, no está depurando la biblioteca.) Por simplicidad, enviamos todo el trabajo por array new y delete a sus homólogos escalares.

Lo siguiente es un test sencillo usando la facilidad MemCheck:

```
//: C02:MemTest.cpp
//{L} MemCheck
// Test of MemCheck system.
#include <iostream>
#include <vector>
#include <cstring>
#include "MemCheck.h"   // Must appear last!
using namespace std;

class Foo {
  char* s;
public:
  Foo(const char*s ) {
    this->s = new char[strlen(s) + 1];
    strcpy(this->s, s);
  }
  ~Foo() { delete [] s; }
};

int main() {
  MEM_ON();
  cout << "hello" << endl;
  int* p = new int;
  delete p;
  int* q = new int[3];
  delete [] q;
  int* r;
  delete r;
  vector<int> v;
  v.push_back(1);
  Foo s("goodbye");
  MEM_OFF();
} ///:~
```

Este ejemplo verifica que puede usar MemCheck en presencia de streams, contenedores estándar, y clases que asignan memoria en constructores. Los punteros p y q son asignados y desasignados sin ningún problema, pero r no es un puntero de pila válido, así que la salida indica el error como un intento de borrar un puntero desconocido:

hola Asignados 4 bytes en la dirección 0xa010778 (fichero: memtest.cpp, línea: 25) Deleted memory at address 0xa010778 Asignados 12 bytes en la dirección 0xa010778 (fichero: memtest.cpp, línea: 27) Memoria borrada en la dirección 0xa010778 Intento de borrar puntero desconocido: 0x1 Asignados 8 bytes en la dirección 0xa0108c0 (fichero: memtest.cpp, línea: 14) Memoria borrada en la dirección 0xa0108c0 ¡No hay agujeros de memoria de usuario!

A causa de la llamada a MEM_OFF( ), no se procesan posteriores llamadas al operador delete( ) por vector o ostream. Todavía podría conseguir algunas llamadas a delete realizadas dsede reasignaciones por los contenedores.

Si llama a TRACE_OFF( ) al principio del programa, la salida es

Hola Intento de borrar puntero desconocido: 0x1 ¡No hay agujeros de memoria de usuario!

3.4. Resumen

Muchos de los dolores de cabeza de la ingeniería del software pueden ser evitados reflexionando sobre lo que está haciendo. Probablemente ha estado usando aserciones mentales cuando ha navegado por sus blucles y funciones, incluso si no ha usado rutinariamente la macro assert( ). Si usa assert( ), encontrará errores lógicos más pronto y acabará con código más legible también. Recuerde usar solo aserciones para invariantes, aunque, no para el manejo de error en tiempo de ejecución.

Nada le dará más tranquilidad que código probado rigurosamente. Si ha sido un lío en el pasado, use un framework automatizado, como el que hemos presentado aquí, para integrar la rutina de pruebas en su trabajo diario. Usted (¡y sus usarios!) estarán contentos de que lo haga.

3.5. Ejercicios

Las soluciones para ejercicios seleccionados pueden encontrarse en el documento electrónico Pensar en C++ Volumen 2 Guía de Soluciones Comentadas disponible por una pequeña cuota en www.MindView.net.

1. Escriba un programa de prueba usando el Framework TestSuite para la clase estándar vector que prueba rigurosamente prueba las siguientes funciones con un vector de enteros: push_back( ) (añade un elemento al final del vector) front( ) (devuelve el primer elemento en el vector), back( ) (devuelve el último elemento en el vector), pop_back( ) (elimina el último elemento sin devolverlo), at( ) (devuelve el elemento en una posición específica), y size( ) (devuelve el número de elementos). Asegúrese de verificar que vector::at( ) lanza una excepción std::out_of_range si el índice facilitado está fuera de rango.

2. Supóngase que le piden desarrollar un clase llamada Rational que da soporte a números racionales (fracciones). La fracción en un objecto Rational debería siempre almacenarse en los términos más bajos, y un denominador de cero es un error. Aquí está una interfaz de ejemplo para esa clase Rational:

```
//: C02:Rational.h {-xo}
#ifndef RATIONAL_H
#define RATIONAL_H
#include <iosfwd>

class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  Rational operator-() const;
  friend Rational operator+(const Rational&,
                            const Rational&);
  friend Rational operator-(const Rational&,
                            const Rational&);
  friend Rational operator*(const Rational&,
                            const Rational&);
  friend Rational operator/(const Rational&,
```

```
                              const Rational&);
    friend std::ostream&
    operator<<(std::ostream&, const Rational&);
    friend std::istream&
    operator>>(std::istream&, Rational&);
    Rational& operator+=(const Rational&);
    Rational& operator-=(const Rational&);
    Rational& operator*=(const Rational&);
    Rational& operator/=(const Rational&);
    friend bool operator<(const Rational&,
                          const Rational&);
    friend bool operator>(const Rational&,
                          const Rational&);
    friend bool operator<=(const Rational&,
                           const Rational&);
    friend bool operator>=(const Rational&,
                           const Rational&);
    friend bool operator==(const Rational&,
                           const Rational&);
    friend bool operator!=(const Rational&,
                           const Rational&);
};
#endif // RATIONAL_H ///:~
```

Escriba una especificación completa para esta clase, incluyendo especificaciones de precondiciones, postcondiciones, y de excepción.

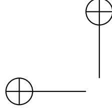
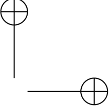
3. Escriba un prueba usando el framework TestSuite que pruebe rigurosamente todas las especificaciones del ejercicio anterior, incluyendo probar las excepciones.

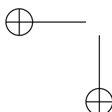
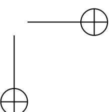
4. Implemente la clase Rational de modo que pase todas las pruebas del ejercicio anterior. Use aserciones sólo para las invariantes.

5. El fichero BuggedSearch.cpp de abajo contiene un función de búsqueda binaria que busca para el rango [pedir, final). Hay algunos errores en el algoritmo. Use las técnicas de seguimiento de este capítulo para depurar la función de búsqueda.

Parte II

La librería Estándar de C++

El C++ Estándar no solo incorpora todas las librerías de Estándar C (con pequeños añadidos y cambios para permitir tipos seguros), también añade sus propias librerías. Estas librerías son mucho más potentes que las de C. La mejora al usarlas es análoga a la que se consigue al cambiar de C a C++.

Esta sección del libro le da una introducción en profundidad a las partes clave de la librería Estándar de C++.

La referencia más completa y también la más oscura para las librerías es el propio Estándar. *The C++ Programming Language, Third Edition (Addison Wesley, 2000)* de Bjarne Stroustrup sigue siendo una referencia fiable tanto para el lenguaje como para la librería. La referencia más aclamada en cuanto a la librería es *The C++ Standard Library: A Tutorial and Reference, by Nicolai Josuttis (Addison Wesley, 1999)*. El objetivo de los capítulos de esta parte del libro es ofrecer un catálogo de descripciones y ejemplos para que disponga de un buen punto de partida para resolver cualquier problema que requiera el uso de las librerías Estándar. Sin embargo, algunas técnicas y temas se usan poco y no se tratan aquí. Si no puede encontrar algo en estos capítulos, mire en los dos libros que se citan anteriormente; esto libro no pretende reemplazarlos, más bien completarlos. En particular, esperamos que después de consular el material de los siguientes capítulos pueda comprender mejor esos libros.

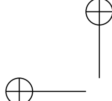
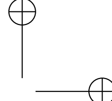
El lector notará que estos capítulos no contienen documentación exhaustiva describiendo cada función o clase del la Librería Estándar C++. Hemos dejado las descripciones completas a otros; en particular a *Dinkumware C/C++ Library Reference* de P.J. Plauger. Esta es una excelente documentación que puede puede ver con un navegador web cada vez que necesite buscar algo. Puede verla on-line o comprarla para verla en local. Contiene una referencia completa para las librerías de C y C++ (de modo que es buena para cualquier cuestión de programación en C/C++ Estándar). La documentación electrónica no sólo es efectiva porque pueda tenerla siempre a mano, sino porque también puede hacer búsquedas electrónicas.

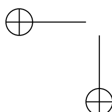
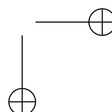
Cuando usted está programando activamente, estos recursos deberían satisfacer sus necesidades de referencias (y puede usarlas para buscar algo de este capítulo que no tenga claro). El Apéndice A incluye referencias adicionales.

El primer capítulo de esta sección introduce la clase `string` del Estándar C++, que es una herramienta potente que simplifica la mayoría de las tareas de procesamiento de texto que podría tener que realizar. Casi cualquier cosas que tenga hecho para cadenas de caracteres en C puede hacerse con una llamada a un método de la clase `string`.

El capítulo 4 cubre la librería `iostreams`, que contiene clases para procesar entrada y salida con ficheros, cadenas, y la consola del sistema.

Aunque el Capítulo 5: «Las plantillas a fondo» no es explícitamente un capítulo de la librería, es una preparación necesaria para los dos siguientes capítulos. En el capítulo 6 examinaremos los algoritmos genéricos que ofrece la librería Estándar C++. Como están implementados con plantillas, esos algoritmos se pueden aplicar a cualquier secuencia de objetos. El Capítulo 7 cubre los contenedores estándar y sus iteradores asociados Vemos los algoritmos primero porque se pueden utilizar usando únicamente arrays y el contenedor `vector` (que vimos en el Volumen 1). También es normal el uso de algoritmos estándar junto con contenedores, y es bueno que le resulten familiares antes de estudiar los contenedores.

4: Las cadenas a fondo

El procesamiento de cadenas de caracteres en C es una de las mayores pérdidas de tiempo. Las cadenas de caracteres requieren que el programador tenga en cuenta las diferencias entre cadenas estáticas y las cadenas creadas en la pila y en el montón, además del hecho que a veces pasa como argumento un char* y a veces hay que copiar el arreglo entero.

Precisamente porque la manipulación de cadenas es muy común, las cadenas de caracteres son una gran fuente de confusiones y errores. Es por ello que la creación de clases de cadenas sigue siendo desde hace años un ejercicio común para programadores novatos. La clase `string` de la biblioteca estándar de C++ resuelve el problema de la manipulación de caracteres de una vez por todas, gestionando la memoria incluso durante las asignaciones y las construcciones de copia. Simplemente no tiene que preocuparse por ello.

Este capítulo[1] examina la clase `string` del Estándar C++; empieza con un vistazo a la composición de las `string` de C++ y como la versión de C++ difiere del tradicional arreglo de caracteres de C. Aprenderá sobre las operaciones y la manipulación usando objetos `string`, y verá como éstas se FIXME[acomodan a la variación] de conjuntos de caracteres y conversión de datos.

Manipular texto es una de las aplicaciones más antiguas de la programación, por eso no resulta sorprendente que las `string` de C++ estén fuertemente inspiradas en las ideas y la terminología que ha usado continuamente en C y otros lenguajes. Conforme vaya aprendiendo sobre los `string` de C++, este hecho se debería ir viendo más claramente. Da igual el lenguaje de programación que escoja, hay tres cosas comunes que querrá hacer con las cadenas:

- Crear o modificar secuencias de caracteres almacenados en una cadena
- Detectar la presencia o ausencia de elementos dentro de la cadena
- Traducir entre diversos esquemas para representar cadenas de caracteres

Verá como cada una de estas tareas se resuelve usando objetos `string` en C++.

4.1. ¿Qué es un `string`?

En C, una cadena es simplemente un arreglo de caracteres que siempre incluye un 0 binario (frecuentemente llamado terminador nulo) como elemento final del

[1] Algunos materiales de este capítulo fueron creados originalmente por Nancy Nicolaisen

arreglo. Existen diferencias significativas entre los `string` de C++ y sus progenitoras en C. Primero, y más importante, los `string` de C++ esconden la implementación física de la secuencia de caracteres que contiene. No debe preocuparse de las dimensiones del arreglo o del terminador nulo. Un `string` también contiene cierta información para uso interno sobre el tamaño y la localización en memoria de los datos. Específicamente, un objeto `string` de C++ conoce su localización en memoria, su contenido, su longitud en caracteres, y la cantidad de caracteres que puede crecer antes de que el objeto `string` deba redimensionar su buffer interno de datos. Las `string` de C++, por tanto, reducen enormemente las probabilidades de cometer uno de los tres errores de programación en C más comunes y destructivos: sobrescribir los límites del arreglo, intentar acceder a un arreglo no inicializado o con valores de puntero incorrectos, y dejar punteros colgando después de que el arreglo deje de ocupar el espacio que estaba ocupando.

La implementación exacta del esquema en memoria para una clase string no esta definida en el estándar C++. Esta arquitectura esta pensada para ser suficientemente flexible para permitir diferentes implementaciones de los fabricantes de compiladores, garantizando igualmente un comportamiento predecible por los usuarios. En particular, las condiciones exactas de cómo situar el almacenamiento para alojar los datos para un objeto `string` no están definidas. FIXME: Las reglas de alojamiento de un `string` fueron formuladas para permitir, pero no requerir, una implementación con referencias múltiples, pero dependiendo de la implementación usar referencias múltiples sin variar la semántica. Por decirlo de otra manera, en C, todos los arreglos de char ocupan una única región física de memoria. En C++, los objetos `string` individuales pueden o no ocupar regiones físicas únicas de memoria, pero si su conjunto de referencias evita almacenar copias duplicadas de datos, los objetos individuales deben parecer y actuar como si tuvieran sus propias regiones únicas de almacenamiento.

```
//: C03:StringStorage.h
#ifndef STRINGSTORAGE_H
#define STRINGSTORAGE_H
#include <iostream>
#include <string>
#include "../TestSuite/Test.h"
using std::cout;
using std::endl;
using std::string;

class StringStorageTest : public TestSuite::Test {
public:
  void run() {
    string s1("12345");
    // This may copy the first to the second or
    // use reference counting to simulate a copy:
    string s2 = s1;
    test_(s1 == s2);
    // Either way, this statement must ONLY modify s1:
    s1[0] = '6';
    cout << "s1 = " << s1 << endl;  // 62345
    cout << "s2 = " << s2 << endl;  // 12345
    test_(s1 != s2);
  }
};
#endif // STRINGSTORAGE_H ///:~
```

Decimos que cuando una implementación solo hace una sola copia al modificar el `string` usa una estrategia de copiar al escribir. Esta aproximación ahorra tiempo y espacio cuando usamos `string` como parámetros por valor o en otras situaciones de solo lectura.

El uso de referencias múltiples en la implementación de una librería debería ser transparente al usuario de la clase string. Desgraciadamente, esto no es siempre el caso. En programas multihilo, es prácticamente imposible usar implementaciones con múltiples referencias de forma segura[32].[2]

4.2. Operaciones con cadenas

Si ha programado en C, estará acostumbrado a la familia de funciones que leen, escriben, modifican y copian cadenas. Existen dos aspectos poco afortunados en la funciones de la librería estándar de C para manipular cadenas. Primero, hay dos familias pobremente organizadas: el grupo plano, y aquellos que requieren que se les suministre el número de caracteres para ser consideradas en la operación a mano. La lista de funciones en la librería de cadenas de C sorprende al usuario desprevenido con una larga lista de nombres crípticos y mayoritariamente impronunciables. Aunque el tipo y número de argumentos es algo consistente, para usarlas adecuadamente debe estar atento a los detalles de nombres de la función y a los parámetros que le pasas.

La segunda trampa inherente a las herramientas para cadenas del estándar de C es que todas ellas explícitamente confían en la asunción de que cada cadena incluye un terminador nulo. Si por confusión o error el terminador nulo es omitido o sobrescrito, poco se puede hacer para impedir que las funciones de cadena de C manipulen la memoria más allá de los límites del espacio de alojamiento, a veces con resultados desastrosos.

C++ aporta una vasta mejora en cuanto a conveniencia y seguridad de los objetos `string`. Para los propósitos de las actuales operaciones de manipulación, existe el mismo número de funciones que la librería de C, pero gracias a la sobrecarga, la funcionalidad es mucho mayor. Además, con una nomenclatura más sensata y un acertado uso de los argumentos por defecto, estas características se combinan para hacer de la clase `string` mucho más fácil de usar que la biblioteca de funciones de cadena de C.

4.2.1. Añadiendo, insertando y concatenando cadenas

Uno de los aspectos más valiosos y convenientes de los `string` en C++ es que crecen cuando lo necesitan, sin intervención por parte del programador. No solo hace el código de manejo del `string` sea inherentemente mas confiable, además elimina por completo las tediosas funciones "caseras" para controlar los limites del almacenamiento en donde nuestra cadena reside. Por ejemplo, si crea un objeto `string` e inicializa este `string` con 50 copias de "X", y después copia en el 50 copias de "Zowie", el objeto, por sí mismo, readecua suficiente almacenamiento para acomodar el crecimiento de los datos. Quizás en ningún otro lugar es más apreciada esta propiedad que cuando las cadenas manipuladas por su código cambian de tamaño y no sabe cuan grande puede ser este cambio. La función miembro `append()` e `insert()` de `string` reubican de manera transparente el almacenamiento cuando un

[2] Es dificil hacer implementaciones con multiples referencias para trabajar de manera segura en multihilo. (Ver [**?**, **?**]). Ver Capitulo 10 para más información sobre multiples hilos

`string` crece:

```
//: C03:StrSize.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  string bigNews("I saw Elvis in a UFO. ");
  cout << bigNews << endl;
  // How much data have we actually got?
  cout << "Size = " << bigNews.size() << endl;
  // How much can we store without reallocating?
  cout << "Capacity = " << bigNews.capacity() << endl;
  // Insert this string in bigNews immediately
  // before bigNews[1]:
  bigNews.insert(1, " thought I");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = " << bigNews.capacity() << endl;
  // Make sure that there will be this much space
  bigNews.reserve(500);
  // Add this to the end of the string:
  bigNews.append("I've been working too hard.");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = " << bigNews.capacity() << endl;
} ///:~
```

Aquí la salida desde un compilador cualquiera:

```
I saw Elvis in a UFO.
Size = 22
Capacity = 31
I thought I saw Elvis in a UFO.
Size = 32
Capacity = 47
I thought I saw Elvis in a UFO. I've been
working too hard.
Size = 59
Capacity = 511
```

Este ejemplo demuestra que aunque puede ignorar con seguridad muchas de las responsabilidades de reserva y gestión de la memoria que tus `string` ocupan, C++ provee a los `string` con varias herramientas para monitorizar y gestionar su tamaño. Nótese la facilidad con la que hemos cambiado el tamaño de la memoria reservada para los `string`. La función `size()` retorna el numero de caracteres actualmente almacenados en el `string` y es idéntico a la función miembro `lenght()`. La función `capacity()` retorna el tamaño de la memoria subyacente actual, es decir, el número de caracteres que el `string` puede almacenar sin tener que reservar más memoria. La función `reserve()` es una optimización del mecanismo que indica su intención de especificar cierta cantidad de memoria para un futuro uso; `capacity()` siempre retorna un valor al menos tan largo como la ultima llamada a `reserve()`. La función `resize()` añade espacio si el nuevo tamaño es mayor que el tamaño actual del `string`; sino trunca el `string`. (Una sobrecarga de `resize()` puede especificar una adición diferente de caracteres).

La manera exacta en que las funciones miembro de `string` reservan espacio para sus datos depende de la implementación de la librería. Cuando testeamos una implementación con el ejemplo anterior, parece que se hacia una reserva de una palabra de memoria (esto es, un entero) dejando un byte en blanco entre cada una de ellas. Los arquitectos de la clase string se esforzaron para poder mezclar el uso de las cadenas de caracteres de C y los objetos `string`, por lo que es probable por lo que se puede observar en *StrSize.cpp*, en esta implementación en particular, el byte esté añadido para acomodar fácilmente la inserción de un terminador nulo.

4.2.2. Reemplazar caracteres en cadenas

La función `insert()` es particularmente útil por que te evita el tener que estar seguro de que la inserción de caracteres en un `string` no sobrepasa el espacio reservado o sobrescribe los caracteres que inmediatamente siguientes al punto de inserción. El espacio crece y los caracteres existentes se mueven graciosamente para acomodar a los nuevos elementos. A veces, puede que no sea esto exactamente lo que quiere. Si quiere que el tamaño del `string` permanezca sin cambios, use la función `replace()` para sobrescribir los caracteres. Existe un número de versiones sobrecargadas de `replace()`, pero la más simple toma tres argumentos: un entero indicando donde empezar en el `string`, un entero indicando cuantos caracteres para eliminar del `string` original, y el `string` con el que reemplazaremos (que puede ser diferente en numero de caracteres que la cantidad eliminada). Aquí un ejemplo simple:

```
//: C03:StringReplace.cpp
// Simple find-and-replace in strings.
#include <cassert>
#include <string>
using namespace std;

int main() {
  string s("A piece of text");
  string tag("$tag$");
  s.insert(8, tag + ' ');
  assert(s == "A piece $tag$ of text");
  int start = s.find(tag);
  assert(start == 8);
  assert(tag.size() == 5);
  s.replace(start, tag.size(), "hello there");
  assert(s == "A piece hello there of text");
} ///:~
```

`Tag` es insertada en `s` (notese que la inserción ocurre *antes* de que el valor indicando el punto de inserción y de que el espacio extra haya sido añadido despues de `Tag`), y entonces es encontrada y reemplazada.

Debería cerciorarse de que ha encontrado algo antes de realizar el `replace()`. En los ejemplos anteriores se reemplaza con un char*, pero existe una versión sobrecargada que reemplaza con un `string`. Aqui hay un ejempl más completo de demostración de `replace()`:

```
//: C03:Replace.cpp
#include <cassert>
```

```
#include <cstddef>  // For size_t
#include <string>
using namespace std;

void replaceChars(string& modifyMe,
  const string& findMe, const string& newChars) {
  // Look in modifyMe for the "find string"
  // starting at position 0:
  size_t i = modifyMe.find(findMe, 0);
  // Did we find the string to replace?
  if(i != string::npos)
    // Replace the find string with newChars:
    modifyMe.replace(i, findMe.size(), newChars);
}

int main() {
  string bigNews = "I thought I saw Elvis in a UFO. "
                   "I have been working too hard.";
  string replacement("wig");
  string findMe("UFO");
  // Find "UFO" in bigNews and overwrite it:
  replaceChars(bigNews, findMe, replacement);
  assert(bigNews == "I thought I saw Elvis in a "
         "wig. I have been working too hard.");
} ///:~
```

Si `replace()` no encuentra la cadena buscada, retorna un `string::npos`. El dato miembro `npos` es una constante estatica de la clase `string` que representa una posición de carácter que no existe[33]. [3]

A diferencia de `insert()`, `replace()` no aumentará el espacio de alamcenamiento de `string` si copia nuevos caracteres en el medio de una serie de elementos de array existentes. Sin embargo, sí que cerecerá su espacio si es necesario, por ejemplo, cuando hace un *"reemplazamiento"* que pueda expandir el `string` más allá del final de la memoria reservada actual. Aquí un ejemplo:

```
//: C03:ReplaceAndGrow.cpp
#include <cassert>
#include <string>
using namespace std;

int main() {
  string bigNews("I have been working the grave.");
  string replacement("yard shift.");
  // The first argument says "replace chars
  // beyond the end of the existing string":
  bigNews.replace(bigNews.size() - 1,
    replacement.size(), replacement);
  assert(bigNews == "I have been working the "
         "graveyard shift.");
} ///:~
```

[3] Es una abrviación de *"no position"*, y su valor más alto puede ser representado por el ubicador de `string` `size_type` (`std::size_t` por defecto).

La llamada a `replace()` empieza *"reemplazando"* más allá del final del array existente, que es equivalente a la operación `append()`. Nótese que en este ejemplo `replace()` expande el array coherentemente.

Puede que haya estado buscando a través del capítulo; intentando hacer algo relativamente fácil como reemplazar todas las ocurrencias de un carácter con diferentes caracteres. Al buscar el material previo sobre reemplazar, puede que haya encontrado la respuesta, pero entonces ha empezaro viendo grupos de caracteres y contadores y otras cosas que parecen un poco demasiado complejas. ¿No tiene `string` una manera para reemplazar un carácter con otro simplemente?

Puede escribir fácilmente cada funcin usando las funciones miembro `find()` y `replace()` como se muestra acontinuacion.

```
//: C03:ReplaceAll.h
#ifndef REPLACEALL_H
#define REPLACEALL_H
#include <string>

std::string& replaceAll(std::string& context,
  const std::string& from, const std::string& to);
#endif // REPLACEALL_H ///:~
```

```
//: C03:ReplaceAll.cpp {O}
#include <cstddef>
#include "ReplaceAll.h"
using namespace std;

string& replaceAll(string& context, const string& from,
  const string& to) {
  size_t lookHere = 0;
  size_t foundHere;
  while((foundHere = context.find(from, lookHere))
    != string::npos) {
    context.replace(foundHere, from.size(), to);
    lookHere = foundHere + to.size();
  }
  return context;
} ///:~
```

La versión de `find()` usada aquí toma como segundo argumento la posición donde empezar a buscar y retorna `string::npos` si no lo encuentra. Es importante avanzar en la posición contenida por la variable `lookHere` pasada como subcadena, en caso de que `from` es una subcadena de `to`. El siguiente programa comprueba la funcion `replaceAll()`:

```
//: C03:ReplaceAllTest.cpp
//{L} ReplaceAll
#include <cassert>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
using namespace std;
```

```
int main() {
  string text = "a man, a plan, a canal, Panama";
  replaceAll(text, "an", "XXX");
  assert(text == "a mXXX, a plXXX, a cXXXal, PXXXama");
} ///:~
```

Como puede comprobar, la clase `string` por ella sola no resuelve todos los posibles problemas. Muchas soluciones se han dejado en los algoritmos de la librería estándar[4] por que la clase `string` puede parece justamente como una secuencia STL(gracias a los iteradores descritos antes). Todos los algoritmos genéricos funcionan en un *"rango"* de elementos dentro de un contenedor. Generalmente este rango es justamente desde el principio del contenedor hasta el final. Un objeto `string` se parece a un contenedor de caracteres: para obtener el principio de este rango use `string::begin()`, y para obtener el final del rango use `string::end()`. El siguiente ejemplomuestra el uso del algoritmo `replace()` para reemplazar todas las instancias de un determinado carácter *"X"* con *"Y"*

```
//: C03:StringCharReplace.cpp
#include <algorithm>
#include <cassert>
#include <string>
using namespace std;

int main() {
  string s("aaaXaaaXXaaXXXaXXXXaaa");
  replace(s.begin(), s.end(), 'X', 'Y');
  assert(s == "aaaYaaaYYaaYYYaYYYYaaa");
} ///:~
```

Nótese que esta función `replace()` no es llamada como función miembro de `string`. Además, a diferencia de la función `string::replace()`, que solo realiza un reemplazo, el algoritmo `replace()` reemplaza todas las instancias de un carácter con otro.

El algoritmo `replace()` solo funciona con objetos individuales (en este caso, objetos char) y no reemplazará arreglos constantes o objetos `string`. Desde que un `string` se copmporta como una secuencia STL, un conjunto de algoritmos pueden serle aplicados, que resolverán otros problemas que las funciones miembro de `string` no resuelven.

4.2.3. Concatenación usando operadores no-miembro sobrecargados

Uno de los descubrimientos más deliciosos que esperan al programador de C que está aprendiendo sobre el manejo de cadenas en C++, es lo simple que es combinar y añadir `string` usando los operadores `operator+` y `operator+=`. Estos operadores hacen combinaciones de cadenas sintacticamente parecidas a la suma de datos numéricos:

```
//: C03:AddStrings.cpp
```

[4] Descrito en profundidad en el Capítulo 6.

```
#include <string>
#include <cassert>
using namespace std;

int main() {
  string s1("This ");
  string s2("That ");
  string s3("The other ");
  // operator+ concatenates strings
  s1 = s1 + s2;
  assert(s1 == "This That ");
  // Another way to concatenates strings
  s1 += s3;
  assert(s1 == "This That The other ");
  // You can index the string on the right
  s1 += s3 + s3[4] + "ooh lala";
  assert(s1 == "This That The other The other oooh lala");
} ///:~
```

Usar los operadores `operator+` y `operator+=` es una manera flexible y conveniente de combinar los datos de las cadenas. En la parte derecha de la sentencia, puede usar casi cualquier tipo que evalúe a un grupo de uno o más caracteres.

4.3. Buscar en cadenas

La familia de funciones miembro de `string` `find` localiza un carácter o grupo de caracteres en una cadena dada. Aquí los miembros de la familia `find()` y su uso general:

Función miembro de búsqueda en un `string`

¿Qué/Cómo lo encuentra?

`find()`

Busca en un `string` un carácter determinado o un grupo de caracteres y retorna la posición de inicio de la primera ocurrencia o `npos` si ha sido encontrado.

`find_first_of()`

Busca en un `string` y retorna la posición de la primera ocurrencia de cualquier carácter en un grupo especifico. Si no encuentra ocurrencias, retorna `npos`.

`find_last_of()`

Busca en un `string` y retorna la posición de la última ocurrencia de cualquier carácter en un grupo específico. Si no encuentra ocurrencias, retorna `npos`.

`find_first_not_of( )`

Busca en un `string` y retorna la posición de la primera ocurrencia que no pertenece a un grupo específico. Si no encontramos ningún elemento, retorna un `npos`

`find_last_not_of( )`

Busca en un `string` y retorna la posición del elemento con el indice mayor que no pertenece a un grupo específico. Si no encontramos ningún elemento, retorna un `npos`

`rfind()`

Busca en un `string`, desde el final hasta el origen, un carácter o grupo de caracteres y retorna la posición inicial de la ocurrencia si se ha encontrado alguna. Si no encuentra ocurrencias, retorna `npos`.

El uso más simple de `find()`, busca uno o más caracteres en un `string`. La versión sobrecargada de `find()` toma un parámetro que especifica el/los carácter(es) que buscar y opcionalmente un parámetro que dice donde empezar a buscar en el `string` la primera ocurrencia. (Por defecto la posición de incio es 0). Insertando la llamada a la función `find()` dentro de un bucle puede buscar fácilmente todas las ocurrencias de un carácter dado o un grupo de caracteres dentro de un `string`.

El siguiente programa usa el método del Tamiz de Eratostenes para hallar los números primos menores de 50. Este método empieza con el número 2, marca todos los subsecuentes múltiplos de 2 ya que no son primos, y repite el proceso para el siguiente candidato a primo. El constructor de sieveTest inicializa sieveChars poniendo el tamaño inicial del arreglo de carácter y escribiendo el valor *'P'* para cada miembro.

```
//: C03:Sieve.h
#ifndef SIEVE_H
#define SIEVE_H
#include <cmath>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::sqrt;
using std::string;

class SieveTest : public TestSuite::Test {
  string sieveChars;
public:
  // Create a 50 char string and set each
  // element to 'P' for Prime:
  SieveTest() : sieveChars(50, 'P') {}
  void run() {
    findPrimes();
    testPrimes();
  }
  bool isPrime(int p) {
    if(p == 0 || p == 1) return false;
    int root = int(sqrt(double(p)));
    for(int i = 2; i <= root; ++i)
      if(p % i == 0) return false;
    return true;
  }
  void findPrimes() {
    // By definition neither 0 nor 1 is prime.
    // Change these elements to "N" for Not Prime:
    sieveChars.replace(0, 2, "NN");
    // Walk through the array:
    size_t sieveSize = sieveChars.size();
    int root = int(sqrt(double(sieveSize)));
    for(int i = 2; i <= root; ++i)
      // Find all the multiples:
      for(size_t factor = 2; factor * i < sieveSize;
           ++factor)
        sieveChars[factor * i] = 'N';
```

```
  }
  void testPrimes() {
    size_t i = sieveChars.find('P');
    while(i != string::npos) {
      test_(isPrime(i++));
      i = sieveChars.find('P', i);
    }
    i = sieveChars.find_first_not_of('P');
    while(i != string::npos) {
      test_(!isPrime(i++));
      i = sieveChars.find_first_not_of('P', i);
    }
  }
};
#endif // SIEVE_H ///:~
```

```
//: C03:Sieve.cpp
//{L} ../TestSuite/Test
#include "Sieve.h"

int main() {
  SieveTest t;
  t.run();
  return t.report();
} ///:~
```

La función `find()` puede recorrer el `string`, detectando múltiples ocurrencias de un carácter o un grupo de caracteres, y `find_first_not_of()` encuentra otros caracteres o subcadenas.

No existen funciones en la clase `string` para cambiar entre mayúsculas/minúsculas en una cadena, pero puede crear esa función fácilmente usando la función de la libreria estándar de C `toupper()` y `tolower()`, que cambian los caracteres entre mayúsculas/minúsculas de uno en uno. El ejemplo siguiente ilustra una búsqueda sensible a mayúsculas/minúsculas.

```
//: C03:Find.h
#ifndef FIND_H
#define FIND_H
#include <cctype>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::tolower;
using std::toupper;

// Make an uppercase copy of s
inline string upperCase(const string& s) {
  string upper(s);
  for(size_t i = 0; i < s.length(); ++i)
    upper[i] = toupper(upper[i]);
  return upper;
```

```
}

// Make a lowercase copy of s
inline string lowerCase(const string& s) {
  string lower(s);
  for(size_t i = 0; i < s.length(); ++i)
    lower[i] = tolower(lower[i]);
  return lower;
}

class FindTest : public TestSuite::Test {
  string chooseOne;
public:
  FindTest() : chooseOne("Eenie, Meenie, Miney, Mo") {}
  void testUpper() {
    string upper = upperCase(chooseOne);
    const string LOWER = "abcdefghijklmnopqrstuvwxyz";
    test_(upper.find_first_of(LOWER) == string::npos);
  }
  void testLower() {
    string lower = lowerCase(chooseOne);
    const string UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    test_(lower.find_first_of(UPPER) == string::npos);
  }
  void testSearch() {
    // Case sensitive search
    size_t i = chooseOne.find("een");
    test_(i == 8);
    // Search lowercase:
    string test = lowerCase(chooseOne);
    i = test.find("een");
    test_(i == 0);
    i = test.find("een", ++i);
    test_(i == 8);
    i = test.find("een", ++i);
    test_(i == string::npos);
    // Search uppercase:
    test = upperCase(chooseOne);
    i = test.find("EEN");
    test_(i == 0);
    i = test.find("EEN", ++i);
    test_(i == 8);
    i = test.find("EEN", ++i);
    test_(i == string::npos);
  }
  void run() {
    testUpper();
    testLower();
    testSearch();
  }
};
#endif // FIND_H ///:~
```

```
//: C03:Find.cpp
//{L} ../TestSuite/Test
#include "Find.h"
```

```
#include "../TestSuite/Test.h"

int main() {
  FindTest t;
  t.run();
  return t.report();
} ///:~
```

Tanto las funciones `upperCase()` como `lowerCase()` siguen la misma forma: hacen una copia de la cadena argumento y cambian entre mayúsculas/minúsculas. El programa Find.cpp no es la mejor solución para el problema para las mayúsculas/minúsculas, por lo que lo revisitaremos cuando examinemos la comparación entre cadenas.

4.3.1. Busqueda inversa

Si necesita buscar en una cadena desde el final hasta el principio (para encontrar datos en orden *"último entra / primero sale"*), puede usar la función miembro de `string` `rfind()`.

```
//: C03:Rparse.h
#ifndef RPARSE_H
#define RPARSE_H
#include <cstddef>
#include <string>
#include <vector>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::vector;

class RparseTest : public TestSuite::Test {
  // To store the words:
  vector<string> strings;
public:
  void parseForData() {
    // The ';' characters will be delimiters
    string s("now.;sense;make;to;going;is;This");
    // The last element of the string:
    int last = s.size();
    // The beginning of the current word:
    size_t current = s.rfind(';');
    // Walk backward through the string:
    while(current != string::npos) {
      // Push each word into the vector.
      // Current is incremented before copying
      // to avoid copying the delimiter:
      ++current;
      strings.push_back(s.substr(current, last - current));
      // Back over the delimiter we just found,
      // and set last to the end of the next word:
      current -= 2;
      last = current + 1;
      // Find the next delimiter:
      current = s.rfind(';', current);
```

```
    }
    // Pick up the first word -- it's not
    // preceded by a delimiter:
    strings.push_back(s.substr(0, last));
  }
  void testData() {
    // Test them in the new order:
    test_(strings[0] == "This");
    test_(strings[1] == "is");
    test_(strings[2] == "going");
    test_(strings[3] == "to");
    test_(strings[4] == "make");
    test_(strings[5] == "sense");
    test_(strings[6] == "now.");
    string sentence;
    for(size_t i = 0; i < strings.size() - 1; i++)
      sentence += strings[i] += " ";
    // Manually put last word in to avoid an extra space:
    sentence += strings[strings.size() - 1];
    test_(sentence == "This is going to make sense now.");
  }
  void run() {
    parseForData();
    testData();
  }
};
#endif // RPARSE_H ///:~
```

```
//: C03:Rparse.cpp
//{L} ../TestSuite/Test
#include "Rparse.h"

int main() {
  RparseTest t;
  t.run();
  return t.report();
} ///:~
```

La función miembro de `string` `rfind()` vuelve por la cadena buscando elementos y reporta el indice del arreglo de las coincidencias de caracteres o `string::npos` si no tiene éxito.

4.3.2. Encontrar el primero/último de un conjunto de caracteres

La función miembro `find_first_of( )` y `find_last_of( )` pueden ser convenientemente usadas para crear una pequeña utilidad the ayude a deshechar los espacion en blanco del final e inicio de la cadena. Nótese que no se toca el `string` originar sino que se devuelve una nuevo `string`:

```
//: C03:Trim.h
// General tool to strip spaces from both ends.
```

```
#ifndef TRIM_H
#define TRIM_H
#include <string>
#include <cstddef>

inline std::string trim(const std::string& s) {
  if(s.length() == 0)
    return s;
  std::size_t beg = s.find_first_not_of(" \a\b\f\n\r\t\v");
  std::size_t end = s.find_last_not_of(" \a\b\f\n\r\t\v");
  if(beg == std::string::npos) // No non-spaces
    return "";
  return std::string(s, beg, end - beg + 1);
}
#endif // TRIM_H ///:~
```

La primera prueba checkea si el string esta vacío; en ese caso, ya no se realizan más test, y se retorna una copia. Nótese que una vez los puntos del final son encontrados, el constructor de `string` construye un nuevo `string` desde el viejo, dándole el contador incial y la longitud.

Las pruebas de una herramienta tan general den ser cuidadosas

```
//: C03:TrimTest.h
#ifndef TRIMTEST_H
#define TRIMTEST_H
#include "Trim.h"
#include "../TestSuite/Test.h"

class TrimTest : public TestSuite::Test {
  enum {NTESTS = 11};
  static std::string s[NTESTS];
public:
  void testTrim() {
    test_(trim(s[0]) == "abcdefghijklmnop");
    test_(trim(s[1]) == "abcdefghijklmnop");
    test_(trim(s[2]) == "abcdefghijklmnop");
    test_(trim(s[3]) == "a");
    test_(trim(s[4]) == "ab");
    test_(trim(s[5]) == "abc");
    test_(trim(s[6]) == "a b c");
    test_(trim(s[7]) == "a b c");
    test_(trim(s[8]) == "a \t b \t c");
    test_(trim(s[9]) == "");
    test_(trim(s[10]) == "");
  }
  void run() {
    testTrim();
  }
};
#endif // TRIMTEST_H ///:~
```

```
//: C03:TrimTest.cpp {O}
#include "TrimTest.h"
```

```
// Initialize static data
std::string TrimTest::s[TrimTest::NTESTS] = {
  " \t abcdefghijklmnop \t ",
  "abcdefghijklmnop \t ",
  " \t abcdefghijklmnop",
  "a", "ab", "abc", "a b c",
  " \t a b c \t ", " \t a \t b \t c \t ",
  "\t \n \r \v \f",
  "" // Must also test the empty string
}; ///:~
```

```
//: C03:TrimTestMain.cpp
//{L} ../TestSuite/Test TrimTest
#include "TrimTest.h"

int main() {
  TrimTest t;
  t.run();
  return t.report();
} ///:~
```

En el arrglo de `string`, puede ver que los arreglos de carácter son automáticamente convertidos a objetos `string`. Este arreglo provee casos para checkear el borrado de espacios en blanco y tabuladores en los extremos, además de asegurar que los espacios y tabuladores no son borrados de la mitad de un `string`.

4.3.3. Borrar caracteres de cadenas

Borrar caracteres es fácil y eficiente con la función miembro `erase()`, que toma dos argumentos: donde empezar a borrar caracteres (que por defecto es 0), y cuantos caracteres borrar (que por defecto es `string::npos`). Si especifica más caracteres que los que quedan en el string, los caracteres restantes se borran igualmente (llamando `erase()` sin argumentos borra todos los caracteres del `string`). A veces es útil abrir un fichero HTML y borrar sus etiquetas y caracteres especiales de manera que tengamos algo aproximadamente igual al texto que obtendríamos en el navegador Web, sólo como un fichero de texto plano. El siguiente ejemplo usa `erase()` para hacer el trabajo:

```
//: C03:HTMLStripper.cpp {RunByHand}
//{L} ReplaceAll
// Filter to remove html tags and markers.
#include <cassert>
#include <cmath>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
#include "../require.h"
using namespace std;
```

```
string& stripHTMLTags(string& s) {
  static bool inTag = false;
  bool done = false;
  while(!done) {
    if(inTag) {
      // The previous line started an HTML tag
      // but didn't finish. Must search for '>'.
      size_t rightPos = s.find('>');
      if(rightPos != string::npos) {
        inTag = false;
        s.erase(0, rightPos + 1);
      }
      else {
        done = true;
        s.erase();
      }
    }
    else {
      // Look for start of tag:
      size_t leftPos = s.find('<');
      if(leftPos != string::npos) {
        // See if tag close is in this line:
        size_t rightPos = s.find('>');
        if(rightPos == string::npos) {
          inTag = done = true;
          s.erase(leftPos);
        }
        else
          s.erase(leftPos, rightPos - leftPos + 1);
      }
      else
        done = true;
    }
  }
  // Remove all special HTML characters
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  replaceAll(s, " ", " ");
  // Etc...
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string s;
  while(getline(in, s))
    if(!stripHTMLTags(s).empty())
      cout << s << endl;
} ///:~
```

Este ejemplo borrará incluso las etiquetas HTML que se extienden a lo largo de varias líneas.[5] Esto se cumple gracias a la bandera estática `inTag`, que evalúa a cierto si el principio de una etiqueta es encontrada, pero la etiqueta de finalización correspondiente no es encontrada en la misma línea. Todas la formas de `erase()` aparecen en la función `stripHTMLFlags()`.[6] La versión de `getline()` que usamos aquí es una función (global) declarada en la cabecera de `string` y es útil porque guarda una línea arbitrariamente larga en su argumento `string`. No necesita preocuparse de las dimensiones de un arreglo cuando trabaja con `istream::getline()`. Nótese que este programa usa la función `replaceAll()` vista antes en este capítulo. En el póximo capitulo, usaremos los flujos de cadena para crear una solución más elegante.

4.3.4. Comparar cadenas

Comparar cadenas es inherentemente diferente a comparar enteros. Los nombres tienen un significado universal y constante. Para evaluar la relación entre las magnitudes de dos cadenas, se necesita hacer una comparación léxica. Una comparación léxica significa que cuando se comprueba un carácter para saber si es *"mayor que"* o *"menor que"* otro carácter, está en realidad comparando la representación numérica de aquellos caracteres tal como están especificados en el orden del conjunto de caracteres que está siendo usado. La ordenación más habitual suele ser la secuencia ASCII, que asigna a los caracteres imprimibles para el lenguaje inglés números en un rango del 32 al 127 decimal. En la codificación ASCII, el primer *"carácter"* en la lista es el espacio, seguido de diversas marcas de puntuación común, y después las letras mayúsculas y minúsculas. Respecto al alfabeto, esto significa que las letras cercanas al principio tienen un valor ASCII menor a aquellos más cercanos al final. Con estos detalles en mente, se vuelve más fácil recordar que cuando una comparació léxica reporta que `s1` es *"mayor que"* `s2`, simplemente significa que cuando fueron comparados, el primer carácter diferente en `s1` estaba atrás en el alfabeto que el carácter en la misma posición en `s2`.

C++ provee varias maneras de comparar cadenas, y cada una tiene ventajas. La más simple de usar son las funciones no-miembro sobrecargadas de operador: `operator==`, `operator!=` `operator>`, `operator<`, `operator>=` y `operator<=`.

```
//: C03:CompStr.h
#ifndef COMPSTR_H
#define COMPSTR_H
#include <string>
#include "../TestSuite/Test.h"
using std::string;

class CompStrTest : public TestSuite::Test {
public:
  void run() {
    // Strings to compare
    string s1("This");
    string s2("That");
    test_(s1 == s1);
    test_(s1 != s2);
```

[5] Para mantener la exposición simple, esta version no maneja etiquetas anidadas, como los comentarios.

[6] Es tentador usar aquí las matemáticas para evitar algunas llamadas a `erase()`, pero como en algunos casos uno de los operandos es `string::npos` (el entero sin signo más grande posible), ocurre un desbordamiento del entero y se cuelga el algoritmo.

```
    test_(s1 > s2);
    test_(s1 >= s2);
    test_(s1 >= s1);
    test_(s2 < s1);
    test_(s2 <= s1);
    test_(s1 <= s1);
  }
};
#endif // COMPSTR_H ///:~
```

```
//: C03:CompStr.cpp
//{L} ../TestSuite/Test
#include "CompStr.h"

int main() {
  CompStrTest t;
  t.run();
  return t.report();
} ///:~
```

Los operadores de comaración sobrecargados son útiles para comparar dos cadenas completas y elementos individuales de una cadena de caracteres.

Nótese en el siguiente ejemplo la flexibilidad de los tipos de argumento ambos lados de los operadores de comparación. Por eficiencia, la clase `string` provee operadores sobrecargados para la comparación directa de objetos `string`, literales de cadena, y punteros a cadenas estilo C sin tener que crear objetos `string` temporales.

```
//: C03:Equivalence.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s2("That"), s1("This");
  // The lvalue is a quoted literal
  // and the rvalue is a string:
  if("That" == s2)
    cout << "A match" << endl;
  // The left operand is a string and the right is
  // a pointer to a C-style null terminated string:
  if(s1 != s2.c_str())
    cout << "No match" << endl;
} ///:~
```

La función `c_str()` retorna un const char* que apunta a una cadena estilo C terminada en nulo, equivalente en contenidos al objeto `string`. Esto se vuelve muy útil cuando se quiere pasar un strin a una función C, como atoi() o cualquiera de las funciones definidas en la cabecera *cstring*. Es un error usar el valor retornado por `c_str()` como un argumento constante en cualquier función.

No encontrará el operador *not* (`!`) o los operadores de comparación lógicos (`&&` y `||`) entre los operadore para `string`. (No encontrará ninguna versión sobrecar-

gada de los operadores de bits de C: &, |, ^, o ~.) Los operadores de conversión no miembros sobrecargados para la clases `string` están limitados a un subconjunto que tiene una aplicación clara y no ambigua para caracteres individuales o grupos de caracteres.

La función miembro `compare()` le ofrece un gran modo de comparación más sofisticado y preciso que el conjunto de operadores nomiembro. Provee versiones sobrecargadas para comparar:

- Dos `string` completos
- Parte de un `string` con un `string` completo
- Partes de dos `string`

```
//: C03:Compare.cpp
// Demonstrates compare() and swap().
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This");
  string second("That");
  assert(first.compare(first) == 0);
  assert(second.compare(second) == 0);
  // Which is lexically greater?
  assert(first.compare(second) > 0);
  assert(second.compare(first) < 0);
  first.swap(second);
  assert(first.compare(second) < 0);
  assert(second.compare(first) > 0);
} ///:~
```

La función `swap()` en este ejemplo hace lo que su nombre implica: cambia el contenido del objeto por el del parámetro. Para comparar un subconjunto de caracteres en un o ambos `string`, añada argumentos que definen donde empezar y cuantos caracteres considerar. Por ejemplo, puede usar las siguientes versiones sobrecargadas de `compare()`:

`s1.compare(s1StartPos, s1NumberChars, s2, s2StartPos, s2NumberChars);`

Aqui un ejemplo:

```
//: C03:Compare2.cpp
// Illustrate overloaded compare().
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This is a day that will live in infamy");
  string second("I don't believe that this is what "
                "I signed up for");
  // Compare "his is" in both strings:
```

```
  assert(first.compare(1, 7, second, 22, 7) == 0);
  // Compare "his is a" to "his is w":
  assert(first.compare(1, 9, second, 22, 9) < 0);
} ///:~
```

Hasta ahora, en los ejemplos, hemos usado la sintaxis de indexación de arrays estilo C para referirnos a un carácter individual en un `string`. C++ provee de una alternativa a la notación `s[n]`: el miembro `at()`. Estos dos mecanismos de indexación producen los mismos resultados si todo va bien:

```
//: C03:StringIndexing.cpp
#include <cassert>
#include <string>
using namespace std;

int main() {
  string s("1234");
  assert(s[1] == '2');
  assert(s.at(1) == '2');
} ///:~
```

Sin embargo, existe una importante diferencia entre `[ ]` y `at()` . Cuando usted intenta referenciar el elemento de un arreglo que esta fuera de sus límites, `at()` tiene la delicadeza de lanzar una excepción, mientras que ordinariamente `[ ]` le dejará a su suerte.

```
//: C03:BadStringIndexing.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s("1234");
  // at() saves you by throwing an exception:
  try {
    s.at(5);
  } catch(exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Los programadores responsables no usarán índices erráticos, pero puede que quiera beneficiarse de la comprobación automática de indices, usando`at()` en el lugar de `[ ]` le da la oportunidad de recuperar diligentemente de las referencias a elementos de un arreglo que no existen. La ejecución de sobre uno de nuestros compiladores le da la siguiente salida: *"invalid string position"*

La función miembro `at()` lanza un objeto de clase `out_of_class`, que deriva finalmente de `std::exception`. Capturando este objeto en un manejador de excepciones, puede tomar las medidas adecuadas como recalcular el índice incorrecto o hacer crecer el arreglo. Usar `string::operator[]( )` no proporciona ningún

tipo de protección y es tan peligroso como el procesado de arreglos de caracteres en C.[37] [7]

4.3.5. Cadenas y rasgos de caracteres

El programa *Find.cpp* anterior en este capítulo nos lleva a hacernos la pregunta obvia: ¿por que la comparación sensible a mayúsculas/minúsculas no es parte de la clase estándar `string`? La respuesta nos brinda un interesante transfondo sobre la verdadera naturaleza de los objetos `string` en C++.

Considere qué significa para un carácter tener "mayúscula/minúscula". El Hebreo escrito, el Farsi, y el Kanji no usan el concepto de "mayúscula/minúscula", con lo que para esas lenguas esta idea carece de significado. Esto daria a entender que si existiera una manera de designar algunos lenguages como "todo mayúsculas" o "todo minúsculas", podriamos diseñar una solución generalizada. Sin embargo, algunos leguajes que emplean el concepto de "mayúscula/minúscula", tambien cambian el significado de caracteres particulares con acentos diacríticos, por ejemplo la cedilla del Español, el circumflexo en Francés y la diéresis en Alemán. Por esta razón, cualquier codificación sensible a mayúsculas que intenta ser comprensiva acaba siendo una pesadilla en su uso.

Aunque tratamos habitualmente el `string` de C++ como una clase, esto no es del todo cierto. El tipo `string` es una especialización de algo más general, la plantilla `basic_string< >`. Observe como está declarada `string` en el fichero de cabecera de C++ estándar.

```
typedef basic_string<char> string;
```

Para comprender la naturaleza de la clase `string`, mire la plantilla `basic_string< >`

```
 template<class charT, class traits = char_traits<charT>, class allocator = allocator<charT> > class basic_string;
```

En el Capítulo 5, examinamos las plantillas con gran detalle (mucho más que en el Capítulo 16 del volúmen 1). Por ahora nótese que el tipo `string` es creada cuando instanciamos la plantilla `basic_string` con char. Dentro de la declaración plantilla `basic_string< >` la línea:

```
class traits = char_traits<charT<,
```

nos dice que el comportamiento de la clase hecha a partir de `basic_string< >` es defineida por una clase basada en la plantilla `char_traits< >`. Así, la plantilla `basic_string< >` produce clases orientadas a `string` que manipulan otros tipos que char (caracteres anchos, por ejemplo). Para hacer esto, la plantilla `char_traits< >` controla el contenido y el comportamiento de la ordenación de una variedad de conjuntos de caracteres usando las funciones de comparación `eq()` (*equal*), `ne()` (*not equal*), y `lt()` (*less than*). Las funciones de comparación de `basic_string< >` confian en esto.

Es por esto por lo que la clase string no incluye funciones miembro sensibles a mayúsculas/minúsculas: eso no esta en la descripción de su trabajo. Para cambiar la forma en que la clase string trata la comparación de caracteres, tiene que suministrar una plantilla `char_traits< >` diferente ya que define el comportamiento

[7] Por las razones de seguridad mencionadas, el C++ *Standards Committee* está considerando una propuesta de redefinición del `string::operator[]` para comportarse de manera idéntica al `string::at()` para C++0x.

individual de las funciones miembro de comparación carácteres.

Puede usar esta información para hacer un nuevo tipo de `string` que ignora las mayúsculas/minúsculas. Primero, definiremos una nueva plantilla no sensible a mayúsculas/minúsculas de `char_traits< >` que hereda de una plantilla existente. Luego, sobrescribiremos sólo los miembros que necesitamos cambiar para hacer la comparación carácter por carácter. (Además de los tres miembros de comparación léxica mencionados antes, daremos una nueva implementación para laspara las funciones de `char_traits` `find()` y `compare()`). Finalmente, haremos un `typedef` de una nueva clase basada en `basic_string`, pero usando nuestra plantilla insensible a mayúsculas/minúsculas, `ichar_traits`, como segundo argumento:

```
//: C03:ichar_traits.h
// Creating your own character traits.
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H
#include <cassert>
#include <cctype>
#include <cmath>
#include <cstddef>
#include <ostream>
#include <string>
using std::allocator;
using std::basic_string;
using std::char_traits;
using std::ostream;
using std::size_t;
using std::string;
using std::toupper;
using std::tolower;

struct ichar_traits : char_traits<char> {
  // We'll only change character-by-
  // character comparison functions
  static bool eq(char c1st, char c2nd) {
    return toupper(c1st) == toupper(c2nd);
  }
  static bool ne(char c1st, char c2nd) {
    return !eq(c1st, c2nd);
  }
  static bool lt(char c1st, char c2nd) {
    return toupper(c1st) < toupper(c2nd);
  }
  static int
  compare(const char* str1, const char* str2, size_t n) {
    for(size_t i = 0; i < n; ++i) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(tolower(*str1) < tolower(*str2))
        return -1;
      else if(tolower(*str1) > tolower(*str2))
        return 1;
      assert(tolower(*str1) == tolower(*str2));
      ++str1; ++str2; // Compare the other chars
    }
    return 0;
```

```
  }
  static const char*
  find(const char* s1, size_t n, char c) {
    while(n-- > 0)
      if(toupper(*s1) == toupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<char, ichar_traits> istring;

inline ostream& operator<<(ostream& os, const istring& s) {
  return os << string(s.c_str(), s.length());
}
#endif // ICHAR_TRAITS_H ///:~
```

Proporcionamos un `typedef` llamado `istring` ya que nuestra clase actuará como un `string` ordinario en todas sus formas, excepto que realizará todas las comparaciones sin respetar las mayúsculas/minúsculas. Por conveniencia, damos un operador sobrecargado `operator<<()` para que pueda imprimir los `istring`. Aque hay un ejemplo:

```
//: C03:ICompare.cpp
#include <cassert>
#include <iostream>
#include "ichar_traits.h"
using namespace std;

int main() {
  // The same letters except for case:
  istring first = "tHis";
  istring second = "ThIS";
  cout << first << endl;
  cout << second << endl;
  assert(first.compare(second) == 0);
  assert(first.find('h') == 1);
  assert(first.find('I') == 2);
  assert(first.find('x') == string::npos);
} ///:~
```

Este es solo un ejemplo de prueba. Para hacer `istring` completamente equivalente a un `string`, deberiamos haber creado las otras funciones necesarias para soportar el nuevo tipo `istring`.

La cabecera <*string*> provee de un `string` ancho [8] gracias al siguiente `typedef`:

`typedef basic_string<wchar_t> wstring;`

El soporte para `string` ancho se revela tambien en los `streams` anchos (`wost-`

[8] (N.del T.) Se refiere a `string` amplio puesto que esta formado por caracteres anchos wchar_t que deben soportar la codificación mas grande que soporte el compilador. Casi siempre esta codificación es *Unicode*, por lo que casi siempre el ancho de wchar_t es 2 bytes

ream en lugar de ostream, tambien definido en <iostream>) y en la especialización de wchar_t de los char_traits en la libreria estándar le da la posibilidad de hacer una version de carácter ancho de ichar_traits

```
//: C03:iwchar_traits.h {-g++}
// Creating your own wide-character traits.
#ifndef IWCHAR_TRAITS_H
#define IWCHAR_TRAITS_H
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cwctype>
#include <ostream>
#include <string>

using std::allocator;
using std::basic_string;
using std::char_traits;
using std::size_t;
using std::towlower;
using std::towupper;
using std::wostream;
using std::wstring;

struct iwchar_traits : char_traits<wchar_t> {
  // We'll only change character-by-
  // character comparison functions
  static bool eq(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) == towupper(c2nd);
  }
  static bool ne(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) != towupper(c2nd);
  }
  static bool lt(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) < towupper(c2nd);
  }
  static int compare(
    const wchar_t* str1, const wchar_t* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(towlower(*str1) < towlower(*str2))
        return -1;
      else if(towlower(*str1) > towlower(*str2))
        return 1;
      assert(towlower(*str1) == towlower(*str2));
      ++str1; ++str2; // Compare the other wchar_ts
    }
    return 0;
  }
  static const wchar_t*
  find(const wchar_t* s1, size_t n, wchar_t c) {
    while(n-- > 0)
      if(towupper(*s1) == towupper(c))
        return s1;
      else
```

```
        ++s1;
      return 0;
    }
};

typedef basic_string<wchar_t, iwchar_traits> iwstring;

inline wostream& operator<<(wostream& os,
  const iwstring& s) {
  return os << wstring(s.c_str(), s.length());
}
#endif // IWCHAR_TRAITS_H  ///:~
```

Como puede ver, esto es principalmente un ejercicio de poner *'w'* en el lugar adecuado del código fuente. El programa de prueba podria ser asi:

```
//: C03:IWCompare.cpp {-g++}
#include <cassert>
#include <iostream>
#include "iwchar_traits.h"
using namespace std;

int main() {
  // The same letters except for case:
  iwstring wfirst = L"tHis";
  iwstring wsecond = L"ThIS";
  wcout << wfirst << endl;
  wcout << wsecond << endl;
  assert(wfirst.compare(wsecond) == 0);
  assert(wfirst.find('h') == 1);
  assert(wfirst.find('I') == 2);
  assert(wfirst.find('x') == wstring::npos);
} ///:~
```

Desgraciadamente, todavia algunos compiladores siguen sin ofrecer un soporte robusto para caracteres anchos.

4.4. Una aplicación con cadenas

Si ha observado atentamente los códigos de ejemplo de este libro, habrá observado que ciertos elementos en los comentarios envuelven el código. Son usados por un programa en Python que escribió Bruce para extraer el código en ficheros y configurar makefiles para construir el código. Por ejemplo, una doble barra segida de dos puntos en el comienzo de una línea denota la primera línea de un fichero de código . El resto de la línea contiene información describiendo el nombre del fichero y su locaización y cuando deberia ser solo compilado en vez constituir un fichero ejecutable. Por ejemplo, la primera línea del programa anterior contiene la cadena *C03:IWCompare.cpp*, indicando que el fichero *IWCompare.cpp* deberia ser extraido en el directorio C03.

La última línea del fichero fuente contiene una triple barra seguida de dos puntos y un signo "~". Es la primera línea tiene una exclamación inmediatamente después de los dos puntos, la primera y la última línea del código fuente no son para ser ex-

traídas en un fichero (solo es para ficheros solo de datos). (Si se está preguntando por que evitamos mostrar estos elementos, es por que no queremos romper el extractor de código cuando lo aplicamos al texto del libro!).

El programa en Python de Bruce hace muchas más cosas que simplemente extraer el código. Si el elemento *"{O}"* sigue al nombre del fichero, su entrada en el makefile solo será configurada para compilar y no para enlazarla en un ejecutable. (El Test Framework en el Capítulo 2 está contruida de esta manera). Para enlazar un fichero con otro fuente de ejemplo, el fichero fuente del ejecutable objetivo contendrá una directiva *"{L}"*, como aquí:

`//{L} ../TestSuite/Test`

Esta sección le presentará un programa para extraer todo el código para que pueda compilarlo e inspeccionarlo manualmente. Puede usar este programa para extraer todo el codigo de este libro salvando el fichero como un fichero de texto[9] (llamémosle TICV2.txt)y ejecutando algo como la siguiente línea de comandos: `C:> extractCode TICV2.txt /TheCode`

Este comando lee el fichero de texto TICV2.txt y escribe todos los archivos de código fuente en subdirectorios bajo el definido */TheCode*. El arbol de directorios se mostrará como sigue:

`TheCode/ C0B/ C01/ C02/ C03/ C04/ C05/ C06/ C07/ C08/ C09/ C10/ C11/ TestSuite/`

Los ficheros de código fuente que contienen los ejemplos de cada capítulo estarán en el correspondiente directorio.

Aquí está el programa:

```
//: C03:ExtractCode.cpp {-edg} {RunByHand}
// Extracts code from text.
#include <cassert>
#include <cstddef>
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Legacy non-standard C header for mkdir()
#if defined(__GNUC__) || defined(__MWERKS__)
#include <sys/stat.h>
#elif defined(__BORLANDC__) || defined(_MSC_VER) \
  || defined(__DMC__)
#include <direct.h>
#else
#error Compiler not supported
#endif

// Check to see if directory exists
// by attempting to open a new file
// for output within it.
bool exists(string fname) {
```

[9] Esté alerta porque algunas versiones de Microsoft Word que substituyen erroneamente los caracteres con comilla simple con un carácter ASCII cuando salva el documento como texto, causan un error de compilación. No tenemos idea de porqué pasa esto. Simplemente reemplace el carácter manualmente con un apóstrofe.

```
  size_t len = fname.length();
  if(fname[len-1] != '/' && fname[len-1] != '\\')
    fname.append("/");
  fname.append("000.tmp");
  ofstream outf(fname.c_str());
  bool existFlag = outf;
  if(outf) {
    outf.close();
    remove(fname.c_str());
  }
  return existFlag;
}

int main(int argc, char* argv[]) {
  // See if input file name provided
  if(argc == 1) {
    cerr << "usage: extractCode file [dir]" << endl;
    exit(EXIT_FAILURE);
  }
  // See if input file exists
  ifstream inf(argv[1]);
  if(!inf) {
    cerr << "error opening file: " << argv[1] << endl;
    exit(EXIT_FAILURE);
  }
  // Check for optional output directory
  string root("./");  // current is default
  if(argc == 3) {
    // See if output directory exists
    root = argv[2];
    if(!exists(root)) {
      cerr << "no such directory: " << root << endl;
      exit(EXIT_FAILURE);
    }
    size_t rootLen = root.length();
    if(root[rootLen-1] != '/' && root[rootLen-1] != '\\')
      root.append("/");
  }
  // Read input file line by line
  // checking for code delimiters
  string line;
  bool inCode = false;
  bool printDelims = true;
  ofstream outf;
  while(getline(inf, line)) {
    size_t findDelim = line.find("//" "/:~");
    if(findDelim != string::npos) {
      // Output last line and close file
      if(!inCode) {
        cerr << "Lines out of order" << endl;
        exit(EXIT_FAILURE);
      }
      assert(outf);
      if(printDelims)
        outf << line << endl;
      outf.close();
      inCode = false;
      printDelims = true;
```

```
      } else {
        findDelim = line.find("//" ":");
        if(findDelim == 0) {
          // Check for '!' directive
          if(line[3] == '!') {
            printDelims = false;
            ++findDelim;  // To skip '!' for next search
          }
          // Extract subdirectory name, if any
          size_t startOfSubdir =
            line.find_first_not_of(" \t", findDelim+3);
          findDelim = line.find(':', startOfSubdir);
          if(findDelim == string::npos) {
            cerr << "missing filename information\n" << endl;
            exit(EXIT_FAILURE);
          }
          string subdir;
          if(findDelim > startOfSubdir)
            subdir = line.substr(startOfSubdir,
                                 findDelim - startOfSubdir);
          // Extract file name (better be one!)
          size_t startOfFile = findDelim + 1;
          size_t endOfFile =
            line.find_first_of(" \t", startOfFile);
          if(endOfFile == startOfFile) {
            cerr << "missing filename" << endl;
            exit(EXIT_FAILURE);
          }
          // We have all the pieces; build fullPath name
          string fullPath(root);
          if(subdir.length() > 0)
            fullPath.append(subdir).append("/");
          assert(fullPath[fullPath.length()-1] == '/');
          if(!exists(fullPath))
#if defined(__GNUC__) || defined(__MWERKS__)
            mkdir(fullPath.c_str(), 0);  // Create subdir
#else
            mkdir(fullPath.c_str());  // Create subdir
#endif
          fullPath.append(line.substr(startOfFile,
                          endOfFile - startOfFile));
          outf.open(fullPath.c_str());
          if(!outf) {
            cerr << "error opening " << fullPath
                 << " for output" << endl;
            exit(EXIT_FAILURE);
          }
          inCode = true;
          cout << "Processing " << fullPath << endl;
          if(printDelims)
            outf << line << endl;
        }
        else if(inCode) {
          assert(outf);
          outf << line << endl;  // Output middle code line
        }
      }
    }
```

```
  exit(EXIT_SUCCESS);
} ///:~
```

Primero observará algunas directivas de compilación condicionales. La función `mkdir()`, que crea un directorio en el sistema de ficheros, se define por el estándar POSIX[10] en la cabecera (<direct.h>). La respectiva signatura de `mkdir()` también difiere: POSIX especifica dos argumentos, las viejas versiones sólo uno. Por esta razón, existe más de una directiva de compilación condicional después en el programa para elegir la llamada correcta a `mkdir()`. Normalmente no usamos compilaciones condicionales en los ejemplos de este libro, pero en este programa en particular es demasiado útil para no poner un poco de trabajo extra dentro, ya que puede usarse para extraer todo el código con él.

La función `exists()` en *ExtractCode.cpp* prueba que un directorio existe abriendo un fiechero temporal en él. Si la obertura falla, el directorio no existe. Borre el fichero enviando su nombre como unchar* a `std::remove()`.

El programa principal valida los argumentos de la línea de comandos y después lee el fichero de entrada línea por línea, mirando por los delimitadores especiales de código fuente. La bandera booleana `inCode` indica que el programa esta en el medio de un fichero fuente, así que las lineas deben ser extraídas. La bandera `printDelims` será verdadero si el elemento de obertura no está seguido de un signo de exclamanción; si no la primera y la última línea no son escritas. Es importante comprobar el último delimitador primero, por que el elemnto inicial es un subconjuntom y buscando por el elemento inicial debería retornar cierto en ambos casos. Si encontramos el elemento final, verificamos que estamos en el medio del procesamiento de un fichero fuente; sino, algo va mal con la manera en que los delimitadores han sido colocados en el fichero de texto. Si `inCode` es verdadero, todo está bien, y escribiremos (opcionalmente) la última linea y cerraremos el fichero. Cuando el elemento de obertura se encuentra, procesamos el directorio y el nombre del fichero y abrimos el fichero. Las siguientes funciones relacionadas con `string` fueron usadas en este ejemplo: `length( )`, `append( )`, `getline( )`, `find( )` (dos versiones), `find_first_not_of( )`, `substr( )`, `find_first_of( )`, `c_str( )`, y, por supuesto, `operator<<( )`

4.5. Resumen

Los objetos `string` proporcionan a los desarrolladores un gran número de ventajas sobre sus contrapartidas en C. La mayoria de veces, la clase `string` hacen a las cadenas con punteros a caracteres innecesarios. Esto elimina por completo una clase de defectos de software que radican en el uso de punteros no inicializados o con valores incorrectos.

FIXME: Los `string` de C++, de manera transparente y dinámica, hacen crecer el espacio de alamcenamiento para acomodar los cambios de tamaño de los datos de la cadena. Cuando los datos en n `string` crece por encima de los límites de la memoria asignada inicialmente para ello, el objeto `string` hará las llamadas para la gestión de la memoria para obtener el espacio y retornar el espacio al montón. La gestión consistente de la memoria previente lagunas de memoria y tiene el potencial de ser mucho más eficiente que un *"hágalo usted mismo"*.

[10] POSIX, un estándar IEEE, es un *"Portable Operating System Interface"* (Interficie de Sistema Operativo Portable) y es una generalización de muchas de las llamadas a sistema de bajo nivel encontradas en los sistemas UNIX.

Las funciones de la clase `string` proporcionan un sencillo y comprensivo conjunto de herramientas para crear, modificar y buscar en cadenas. Las comparaciones entre `string` siempre son sensibles a mayúsculas/minúsculas, pero usted puede solucionar el problema copiando los datos a una cadena estilo C acabada en nulo y usando funciones no sensibles a mayúsculas/minúsculas, convirtiendo temporalmente los datos contenidos a mayúsculas o minúsculas, o creando una clase `string` sensible que sobreescribe los rasgos de carácter usados para crear un objeto `basic_string`

4.6. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Escriba y pruebe una función que invierta el orden de los caracteres en una cadena.

2. 2. Un palindromo es una palabra o grupo de palabras que tanto hacia delante hacia atrás se leen igual. Por ejemplo *"madam"* o *"wow"*. Escriba un programa que tome un string como argumento desde la línea de comandos y, usando la función del ejercicio anterior, escriba si el string es un palíndromo o no.

3. 3. Haga que el programa del Ejercicio 2 retorne verdadero incluso si las letras simetricas difieren en mayúsculas/minúsculas. Por ejemplo, *"Civic"* debería retornar verdadero aunque la primera letra sea mayúscula.

4. 4. Cambie el programa del Ejercicio 3 para ignorar la puntuación y los espacios también. Por ejemplo *"Able was I, ere I saw Elba."* debería retornar verdadero.

5. 5. Usando las siguientes declaraciones de string y solo char (no literales de cadena o números mágicos):

   ```
   string one("I walked down the canyon with the moving mountain bikers."); string two("The bikers passed by me too close for comfort."); string three("I went hiking instead.");
   ```

   produzca la siguiente frase:

   I moved down the canyon with the mountain bikers. The mountain bikers passed by me too close for comfort. So I went hiking instead.

6. 6. Escriba un programa llamado "reemplazo" que tome tres argumentos de la línea de comandos representando un fichero de texto de entrada, una frase para reemplazar (llámela `from`), y una cadena de reemplazo (llámela `to`). El programa debería escribir un nuevo fichero en la salida estandar con todas las ocurrencias de `from` reemplazadas por `to`.

7. 7. Repetir el ejercicio anterior pero reemplazando todas las instancias pero ignorando las mayúsculas/minúsculas.

8. 8. Haga su programa a partir del Ejercicio 3 tomando un nombre de fichero de la linea de comandos, y despues mostrando todas las palabras que son palíndromos (ignorando las mayúsculas/minúsculas) en el fichero. No intente buscar palabras para palíndromos que son mas largos que una palabra (a diferencia del ejercicio 4).

9. 9. Modifique *HTMLStripper.cpp* para que cuando encuentre una etiqueta, muestre el nombre de la etiqueta, entonces muestre el contenido del fichero entre la etiqueta y la etiqueta de finalización de fichero. Asuma que no existen etiquetas anidadas, y que todas las etiquetas tienen etiquetas de finalizacion (denotadas con </TAGNAME>).

10. 10. Escriba un programa que tome tres argumentos de la línea de comandos (un nombre de fichero y dos cadenas) y muestre en la consola todas la líneas en el fichero que tengan las dos cadenas en la línea, alguna cadena, solo una cadena o ninguna de ellas, basándose en la entreada de un usuario al principio del programa (el usuario elegirá que modo de búsqueda usar). Para todo excepto para la opción "ninguna cadena", destaque la cadena(s) de entrada colocando un asterisco (*) al principio y al final de cada cadena que coincida cuando sea mostrada.

11. 11. Escriba un programa que tome dos argumentos de la linea de comandos (un nombre de fichero y una cadena) y cuente el numero de veces que la cadena esta en el fichero, incluso si es una subcadena (pero ignorando los solapamientos). Por ejemplo, una cadena de entrada de *"ba"* debería coincidir dos veces en la palabra *"basquetball"*, pero la cadena de entrada *"ana"* solo deberia coincidir una vez en *"banana"*. Muestre por la consola el número de veces que la cadena coincide en el fichero, igual que la longitud media de las palabras donde la cadena coincide. (Si la cadena coincide más de una vez en una palabra, cuente solamente la palabra una vez en el cálculo de la media).

12. 12. Escriba un programa que tome un nombre de fichero de la línea de comandos y perfile el uso del carácter, incluyendo la puntuación y los espacios (todos los valores de caracteres desde el 0x21 [33] hasta el 0x7E [126], además del carácter de espacio). Esto es, cuente el numero de ocurrencias para cada carácter en el fichero, después muestre los resultados ordenados secuencialmente (espacio, despues !, ", #, etc.) o por frecuencia descendente o ascendente basado en una entrada de usuario al principio del programa. Para el espacio, muestre la palabra "espacio" en vez del carácter ' '. Una ejecución de ejemplo debe mostrarse como esto:

    Formato secuencial, ascendente o descendente (S/A/D): D t: 526 r: 490 etc.

13. 13. Usando `find()` y `rfind()`, escriba un programa que tome dos argumentos de lánea de comandos (un nombre de fichero y una cadena) y muestre la primera y la última palapra (y sus indices) que no coinciden con la cadena, asi como los indice de la primera y la última instancia de la cadena. Muestre *"No Encontrado"* si alguna de las busquedas fallan.

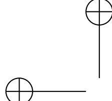
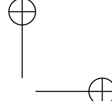
14. 14. Usando la familia de fuciones `find_first_of` (pero no exclusivamente), escriba un programa que borrará todos los caracteres no alfanuméricos excepto los espacios y los puntos de un fichero. Despues convierta a mayúsculas la primera letra que siga a un punto.

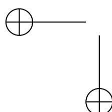
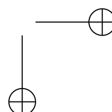
15. 15. Otra vez, usando la familia de funciones `find_first_of`, escriba un programa que acepte un nombre de fichero como argumentod elinea de comandos y después formatee todos los números en un fichero de moneda. Ignore los puntos decimales después del primero despues de un carácter no mumérico, e redondee al

16. 16. Escriba un programa que acepte dos argumentos por línea de comandos (un nombre de fichero y un numero) y mezcle cada paralabra en el fichero cambiando aleatoriamente dos de sus letras el número de veces especificado

en el segundo parametro. (Esto es, si le pasamos 0 a su programa desde la línea de comandos, las palabras no serán mezcladas; si le pasamos un 1, un par de letras aleatoriamente elegidas deben ser cambiadas, para una entrada de 2, dos parejas aleatorias deben ser intercambiadas, etc.).

17. 17. Escriba un programa que acepte un nombre de fichero desde la línea de comandos y muestre el numero de frases (definido como el numero de puntos en el fichero), el número medio de caracteres por frase, y el número total de caracteres en el fichero.

5: Iostreams

Puedes hacer mucho más con el problema general de E/S que simplemente coger el E/S estándar y convertirlo en una clase.

¿No seria genial si pudiera hacer que todos los 'receptáculos' -E/S estándar, ficheros, e incluso boques de memoria- parecieran iguales de manera que solo tuviera que recordar una interficie? Esta es la idea que hay detrás de los `iostreams`. Son mucho más sencillos, seguros, y a veces incluso más eficientes que el conjunto de funciones de la libreria estándar de C `stdio`.

Las clases de `iostream` son generalmente la primera parte de la libreria de C++ que los nuevos programadores de C++ parender a usar. En este capítulo se discute sobre las mejoras que representan los `iostream` sobre las funciones de `stdio` de C y explora el comprotamiento de los ficheros y streams de strings además de los streams de consola.

5.1. ¿Por que `iostream`?

Se debe estar preguntando que hay de malo en la buena y vieja librería de C. ¿Por que no 'incrustar' la libreria de C en una clase y ya está? A veces esta solución es totalmente válida. Por ejemplo, suponga que quiere estar seguro que un fichero representado por un puntero de `stdio FILE` siempre es abierto de forma segura y cerrado correctamente sin tener que confiar en que el usuario se acuerde de llamar a la función `close()`. El siguiente programa es este intento:

```
//: C04:FileClass.h
// stdio files wrapped.
#ifndef FILECLASS_H
#define FILECLASS_H
#include <cstdio>
#include <stdexcept>

class FileClass {
  std::FILE* f;
public:
  struct FileClassError : std::runtime_error {
    FileClassError(const char* msg)
    : std::runtime_error(msg) {}
  };
  FileClass(const char* fname, const char* mode = "r");
  ~FileClass();
  std::FILE* fp();
};
```

```
#endif // FILECLASS_H ///:~
```

Cuando trabaja con ficheros E/S en C, usted trabaja con punteros desnudos a una `struct` de `FILE`, pero esta clase envuelve los punteros y garantiza que es correctamente inicializada y destruida usando el constructor y el destructor. El segundo parámetro del constructor es el modo del fichero, que por defecto es 'r' para 'leer'

Para pedir el valor del puntero para usarlo en las funciones de fichero de E/S, use la función de acceso `fp()`. Aquí están las definiciones de las funciones miembro:

```
//: C04:FileClass.cpp {O}
// FileClass Implementation.
#include "FileClass.h"
#include <cstdlib>
#include <cstdio>
using namespace std;

FileClass::FileClass(const char* fname, const char* mode) {
  if((f = fopen(fname, mode)) == 0)
    throw FileClassError("Error opening file");
}

FileClass::~FileClass() { fclose(f); }

FILE* FileClass::fp() { return f; } ///:~
```

El constructor llama a `fopen()`, tal como se haría normalmente, pero además se asegura que el resultado no es cero, que indica un error al abrir el fichero. Si el fichero no se abre correctamente, se lanza una excepción.

El destructor cierra el fichero, y la función de acceso `fp()` retorna `f`. Este es un ejemplo de uso de `FileClass`:

```
//: C04:FileClassTest.cpp
//{L} FileClass
#include <cstdlib>
#include <iostream>
#include "FileClass.h"
using namespace std;

int main() {
  try {
    FileClass f("FileClassTest.cpp");
    const int BSIZE = 100;
    char buf[BSIZE];
    while(fgets(buf, BSIZE, f.fp()))
      fputs(buf, stdout);
  } catch(FileClass::FileClassError& e) {
    cout << e.what() << endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
} // File automatically closed by destructor
///:~
```

Se crea el objeto FileClass y se usa en llamadas a funciones E/S de fichero normal de C, llamando a fp(). Cuando haya acabado con ella, simplemente olvídese; el fichero será cerrado por el destructor al final del ámbito de la variable.

Incluso teniendo en cuenta que FILE es un puntero privado, no es particularmente seguro porque fp() lo recupera. Ya que el único efecto que parece estar garantizado es la inicialización y la liberación, ¿por que no hacerlo público o usar una struct en su lugar? Nótese que mientras se puede obtener una copia de f usando fp(), no se puede asignar a f -que está completamente bajo el control de la clase. Después de capturar el puntero retornado por fp(), el programador cliente todavía puede asignar a la estructura elementos o incluso cerrarlo, con lo que la seguridad esta en la garantía de un puntero a FILE válido mas que en el correcto contenido de la estructura.

Si quiere completa seguridad, tiene que evitar que el usuario acceda directamente al puntero FILE. Cada una de las versiones de las funciones normales de E/S a ficheros deben ser mostradas como miembros de clase para que todo lo que se pueda hacer desde el acercamiento de C esté disponible en la clase de C++.

```
//: C04:Fullwrap.h
// Completely hidden file IO.
#ifndef FULLWRAP_H
#define FULLWRAP_H
#include <cstddef>
#include <cstdio>
#undef getc
#undef putc
#undef ungetc
using std::size_t;
using std::fpos_t;

class File {
  std::FILE* f;
  std::FILE* F(); // Produces checked pointer to f
public:
  File(); // Create object but don't open file
  File(const char* path, const char* mode = "r");
  ~File();
  int open(const char* path, const char* mode = "r");
  int reopen(const char* path, const char* mode);
  int getc();
  int ungetc(int c);
  int putc(int c);
  int puts(const char* s);
  char* gets(char* s, int n);
  int printf(const char* format, ...);
  size_t read(void* ptr, size_t size, size_t n);
  size_t write(const void* ptr, size_t size, size_t n);
  int eof();
  int close();
  int flush();
  int seek(long offset, int whence);
  int getpos(fpos_t* pos);
  int setpos(const fpos_t* pos);
  long tell();
  void rewind();
```

```
  void setbuf(char* buf);
  int setvbuf(char* buf, int type, size_t sz);
  int error();
  void clearErr();
};
#endif // FULLWRAP_H ///:~
```

Esta clase contiene casi todas las funciones de E/S de fichero de `<cstdio>`. (`v-fprintf()` no esta; se implementa en la función miembro `printf()`)

El fichero tiene el mismo constructor que en el ejemplo anterior, y también tiene un constructor por defecto. El constructor por defecto es importante si se crea un array de objetos `File` o se usa un objeto `File` como miembro de otra clase donde la inicialización no se realiza en el contructor, sino cierto tiempo después de que el objeto envolvente se cree.

El constructor por defecto pone a cero el puntero a `FILE` privado `f`. Pero ahora , antes de cualquier referencia a `f`, el valor debe ser comprobado para asegurarse que no es cero. Esto se consigue con `F()`, que es privado porque está pensado para ser usado solamente por otras funciones miembro. (No queremos dar acceso directo a usuarios a la estructura de `FILE` subyacente en esta clase).

Este acercamiento no es terrible en ningún sentido. Es bastante funcional, y se puede imaginar haciendo clases similares para la E/S estándar (consola) y para los formateos en el core (leer/escribir un trozo de la memoria en vez de un fichero o la consola).

Este bloque de código es el interprete en tiempo de ejecución usado para las listas variables de argumentos. Este es el código que analiza el formato de su cadena en tiempo de ejecución y recoge e interpreta argumentos desde una lista variable de argumentos. Es un problema por cuatro razones:

1. Incluso si solo se usa una fracción de la funcionalidad del interprete, se carga todo en el ejecutable. Luego si quiere usar un `printf("%c", 'x');` , usted tendrá todo el paquete, incluido las partes que imprimen números en coma flotante y cadenas. No hay una opción estándar para reducir el la cantidad de espacio usado por el programa.

2. Como la interpretación pasa en tiempo de ejecución, no se puede evitar un empeoramiento del rendimiento. Esto es frustrante por que toda la información está allí, en el formato de la cadena, en tiempo de compilación, pero no se evalua hasta la ejecución. Por otro lado, si se pudieran analizar los argumentos en el formateo de la cadena durante la compilación, se podrían hacer llamadas directas a funciones que tuvieran el potencial de ser mucho más rápidas que un interprete en tiempo de ejecución (aunque la familia de funciones de printf() acostumbran a estar bastante bien optimizadas).

3. Como el formateo de la cadena no se evalua hasta la ejecución, no se hace una comprobación de errores al compilar. Probalblemente está familiarizado con este problema si ha intentado buscar errores que provienen del uso de un número o tipo de argumentos incorrecto en una sentencia `printf()`. C++ ayuda mucho a encontrar rápidamente errores durante la compilación y hacerle la vida más fácil. Parece una tonteria desechar la seguridad en los tipos de datos para la libreria de E/S, especialmente cuando usamos intensivamente las E/S.

4. Para C++, el más crucial de los problemas es que la familia de funciones de

printf() no es particularmente extensible. Esta realmente diseñada para manejar solo los tipos básicos de datos en C (char, int, float, double, wchar_t, char*, wchar_t*, y void*) y sus variaciones. Debe estar pensando que cada vez que añade una nueva clase, puede añadir funciones sobrecargadas printf() y scanf() (y sus variaciones para ficheros y strings), pero recuerde: las funciones sobrecargadas deben tener diferentes tipos de listas de argumentos, y la familia de funciones de printf() esconde esa información en la cadena formateada y su lista variable de argumentos. Para un lenguage como C++, cuya virtud es que se pueden añadir fácilmente nuevos tipos de datos, esta es una restricción inaceptable.

5.2. `Iostreams` al rescate

Estos problemas dejan claro que la E/S es una de las principales prioridades para la librería de clases estándar de C++. Como 'hello, worlod' es el primer programa que cualquiera escribe en un nuevo lenguaje, y porque la E/S es parte de virtualmente cualquier programa, la librería de E/S en C++ debe ser particularmente fácil de usar. Tembién tiene el reto mucho mayor de acomodar cualquier nueva clase. Por tanto, estas restricciones requieren que esta librería de clases fundamentales tengan un diseño realmente inspirado. Además de ganar en abstracción y claridad en su trabajo con las E/S y el formateo, en este capítulo verá lo potente que puede llegar a ser esta librería de C++.

5.2.1. Insertadores y extractores

Un `stream` es un objeto que transporta y formatea carácteres de un ancho fijo. Puede tener un `stream` de entrada (por medio de los descendientes de la clase `istream`), o un `stream` de salida (con objetos derivados de `ostream`), o un `stream` que hace las dos cosas simultáneamente (con objetos derivados de `iostream`). La librería `iostream` provee tipos diferentes de estas clases: `ifstream`, `ofstream` y `fstream` para ficheros, y `istringstream`, `ostringstream`, y `stringstream` para comunicarese con la clase `string` del estándar C++. Todas estas clases `stream` tiene prácticamente la misma interfaz, por lo que usted puede usar streams de manera uniforme, aunque esté trabajando con un fichero, la E/S estándar, una región de la memoria, o un objeto `string`. La única interfaz que aprenderá también funciona para extensiones añadidas para soportar nuevas clases. Algunas funciones implementan sus comandos de formateo, y algunas funciones leen y escriben caracteres sin formatear.

Las clases `stream` mencionadas antes son actualmente especializaciones de plantillas, muchas como la clase estándar `string` son especializaciones de la plantilla `basic_string`. Las clases básicas en la jerarquia de herencias son mostradas en la siguiente figura: [1]

La clase `ios_base` declara todo aquello que es común a todos los `stream`, independientemente del tipo de carácteres que maneja el `stream`. Estas declaraciones son principalmente constantes y funciones para manejarlas, algunas de ella las verá a durante este capítulo. El resto de clases son plantillas que tienen un tipo de caracter subyacente como parámetro. La clase `istream`, por ejemplo, está definida a continuación:

[1] Explicadas en profundidad en el capítulo 5.

```
typedef basic_istream<char< istream;
```

Todas las clases mencionadas antes estan definidas de manera similar. También hay definiciones de tipo para todas las clases de `stream` usando wchar_t (la anchura de este tipo de carácteres se discute en el Capítulo 3) en lugar de char. Miraremos esto al final de este capítulo. La plantilla `basic_ios` define funciones comunes para la entrada y la salida, pero depende del tipo de carácter subyacente (no vamos a usarlo mucho). La plantilla `basic_istream` define funciones genéricas para la entrada y `basic_ostream` hace lo mismo para la salida. Las clases para ficheros y streams de `strings` introducidas después añaden funcionalidad para sus tipos especificos de `stream`.

En la librería de `iostream`, se han sobrecargado dos operadores para simplificar el uso de `iostreams`. El operador << se denomina frecuentemente instertador para `iostreams`, y el operador >> se denomina frecuentemente extractor.

Los extractores analizan la información esperada por su objeto destino de acuerdo con su tipo. Para ver un ejemplo de esto, puede usar el objeto `cin`, que es el equivalente de `iostream` de stdin en C, esto es, entrada estándar redireccionable. Este objeto viene predefinido cuando usted incluye la cabecera `<iostream>`.

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

Existe un operador sobrecargado >> para cada tipo fundamental de dato. Usted también puede sobrecargar los suyos, como verá más adelante.

Para recuperar el contenido de las variables, puede usar el objeto `cout` (correspondiente con la salida estándar; también existe un objeto `cerr` correspondiente con la salida de error estándar) con el insertador <<:

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

Esto es tedioso y no parece ser un gran avance sobre `printf()`, aparte de la mejora en la comprobación de tipos. Afortunadamente, los insertadores y extracto-

res sobrecargados están diseñados para ser encadenados dentro de expresiones más complejas que son mucho más fáciles de escribir (y leer):

```
  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;
```

Definir insertadores y extractores para sus propias clases es simplemente una cuestion de sobrecargar los operadores asociados para hacer el trabajo correcto, de la siguente manera:

Hacer del primer parámetro una referencia no constante al `stream` (`istream` para la entrada, `ostream` para la salida).

Realizar la operación de insertar/extraer datos hacia/desde el `stream` (procesando los componentes del objeto).

Retornar una referencia al `stream`

El `stream` no debe ser constante porque el procesado de los datos del `stream` cambian el estado del `stream`. Retornando el `stream`, usted permite el encadenado de operaciones en una sentencia individual, como se mostró antes.

Como ejemplo, considere como representar la salida de un objeto `Date` en formato MM-DD-AAAA . El siguiente insertador hace este trabajo:

```
ostream& operator<<(ostream& os, const Date& d) {
  char fillc = os.fill('0');
  os << setw(2) << d.getMonth() << '-'
     << setw(2) << d.getDay() << '-'
     << setw(4) << setfill(fillc) << d.getYear();
  return os;
}
```

Esta función no puede ser miembro de la clase `Date` por que el operando de la izquierda << debe ser el `stream` de salida. La función miembro `fill()` de `ostream` cambia el carácter de relleno usado cuando la anchura del campo de salida, determinada por el manipulador `setw()`, es mayor que el necesitado por los datos. Usamos un caracter '0' ya que los meses anteriores a Octubre mostrarán un cero en primer lugar, como '09' para Septiembre. La funcion `fill()` también retorna el caracter de relleno anterior (que por defecto es un espacio en blanco) para que podamos recuperarlo después con el manipulador `setfill()`. Discutiremos los manipuladores en profundidad más adelante en este capítulo.

Los extractores requieren algo más cuidado porque las cosas pueden ir mal con los datos de entrada. La manera de avisar sobre errores en el `stream` es activar el bit de error del `stream`, como se muestra a continuación:

```
istream& operator>>(istream& is, Date& d) {
  is >> d.month;
  char dash;
  is >> dash;
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.day;
  is >> dash;
```

```
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.year;
  return is;
}
```

Cuando se activa el bit de error en un `stream`, todas las operaciones posteriores serán ignoradas hasta que el `stream` sea devuelto a un estado correcto (explicado brevemente). Esto es porque el código de arriba continua extrayendo incluso is `ios::failbit` está activado. Esta implementación es poco estricta ya que permite espacios en blanco entre los numeros y guiones en la cadena de la fecha (por que el operador >> ignora los espacios en blanco por defecto cuado lee tipos fundamentales). La cadena de fecha a continuación es válida para este extractor:

```
"08-10-2003"
"8-10-2003"
"08 - 10 - 2003"
```

Pero estas no:

```
"A-10-2003" // No alpha characters allowed
"08%10/2003" // Only dashes allowed as a delimiter
```

Discutiremos los estados de los `stream` en mayor profundidad en la sección 'Manejar errores de `stream`' después en este capítulo.

5.2.2. Uso común

Como se ilustraba en el extractor de `Date`, debe estar alerta por las entradas erróneas. Si la entrada produce un valor inesperado, el proceso se tuerce y es difícil de recuperar. Además, por defecto, la entrada formateada está delimitada por espacios en blanco. Considere que ocurre cuando recogemos los fragmentos de código anteriores en un solo programa:

```
//: V2C04:Iosexamp.cpp {RunByHand}
```

y le proporcionamos la siguiente entrada:

```
12 1.4 c this is a test
```

esperamos la misma salida que si le hubieramos proporcionado esto:

```
12
1.4
c
this is a test
```

pero la salida es algo inesperado

```
i = 12
f = 1.4
```

```
c = c
buf = this 0xc
```

Nótese que `buf` solo tiene la primera palabra porque la rutina de entrada busca un espacio que delimite la entrada, que es el que se encuentra después de 'tihs.' Además, si la entrada continua de datos es mayor que el espacio reservado por `buf`, sobrepasamos los limites del buffer.

En la práctica, usualmente deseará obtener la entrada desde programas interactivos, una linea cada vez como secuencia de carácteres, leerla, y después hacer las conversiones necesarias hasta que estén seguras en un buffer. De esta manera no deberá preocuparse por la rutina de entrada fallando por datos inesperados.

Otra consideración es todo el concepto de interfaz de línea de comandos. Esto tenia sentido en el pasado cuando la consola era la única interfaz con la máquina, pero el mundo está cambiando rápidamente hacia otro donde la interfaz gráfica de usuario (GUI) domina. ¿Cual es el sentido de la E/S por consola en este mundo? Esto le da mucho más sentido a ignorar `cin` en general, salvo para ejemplos simples y tests, y hacer los siguientes acercamientos:

1. Si su programa requiere entrada, ¿leer esta entrada desde un fichero? Pronto verá que es remarcablemente fácil usar ficheros con `iostream`. `Iostream` para ficheros todavia funciona perfectamente con una GUI.
2. Leer la entrada sin intentar convertirla, como hemos sugerido. Cuando la entrada es algun sitio donde no podemos arriesgarnos durante la conversión, podemos escanearla de manera segura.
3. La salida es diferente. Si está usando una interfaz gráfica, `cout` no necesariamente funciona, y usted debe mandarlo a un fichero (que es indéntico a mandarlo a un `cout`) o usar los componentes del GUI para mostrar los datos. En cualquier otra situacion, a menudo tiene sentido mandarlo a `cout`. En ambos casos, la funciones de formateo de la salida de `iostream` son muy útiles.

Otra práctica común ahorra tiempo en compilaciones largas. Consideres, por ejemplo, cómo quiere declarar los operadores del stream Date introducidos antes en el capítulo en un fichero de cabecera. Usted solo necesita incluir los prototipos para las funciones, luego no es necesario incluir la cabecera entera de <iostream> en Date.h. La práctica estándar es declarar solo las clases, algo como esto:

```
class ostream;
```

Esta es una vieja tecnica para separar la interfaz de la implementación y a menudo la llaman declaración avanzada(y `ostream` en este punto debe ser considerada un tipo incompleto, ya que la definición de la clase no ha sido vista todavia por el compilador).

Esto con funcionará asi, igualmente, por dos razones:

Las clases stream estan definidas en el espacio de nombres `std`.

Son plantillas.

La declaración correcta debería ser:

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
```

```
    class basic_ostream;
  typedef basic_ostream<char> ostream;
}
```

(Como puede ver, como las clase string, las clases stream usan las clases de rasgos de caracter mencionadas en el Capítulo 3). Como puede ser terriblemente tedioso darle un tipo a todas las clases stream a las que quiere referenciar, el estándar provee una cabecera que lo hace por usted:

```
// Date.h
#include <iosfwd>

class Date {
  friend std::ostream& operator<<(std::ostream&,
                                  const Date&);
  friend std::istream& operator>>(std::istream&, Date&);
  // Etc.
```

5.2.3. Entrada orientada a líneas

Para recoger la entrada de línea en línea, tiene tres opciones:

La función miembre `get()`

La función miembro `getline()`

La función global `getline()` definida en la cabecera <string>

Las primeras dos funciones toman tres parámentros:

Un puntero a un buffer de carácters donde se guarda el resultado.

El tamaño de este buffer (para no sobrepasarlo).

El carácter de finalización, para conocer cuando parar de leer la entrada.

El carácter de finalización tiene un valor por defecto de '\n', que es el que usted usará usualmente. Ambas funciones almacenan un cero en el buffer resultante cuando encuentran el caracter de terminación en la entrada.

Entonces, ¿cual es la diferencia? Sutil pero importante: `get()` se detiene cuando vee el delimitador en el stream de entrada, pero no lo extrae de `stream` de entrada. Entonces, si usted hace otro `get()` usando el mismo delimitador, retornará inmediatamente sin ninguna entrada contenida. (Presumiblemente, en su lugar usará un delimitador diferente en la siguiente sentencia `get()` o una función de entrada diferente.) La función `getline()`, por el contrario, extrae el delimitador del `stream` de entrada, pero tampoco lo almacena en el buffer resultante.

La función `getline()` definida en <string> es conveniente. No es una función miembro, sino una función aislada declarada en el espacio de nombres `std`. Sólo toma dos parámetros que no son por defecto, el `stream` de entrada y el objeto `string` para rellenar. Como su propio nombre dice, lee carácteres hasta que encuentra la primera aparición del delimitador ('\n' por defecto) y consume y descarta el delimitador. La ventaja de esta función es que lo lee dentro del objeto `string`, así que no se tiene que preocuparse del tamaño del buffer.

Generalmente, cuando esta procesando un fichero de texto en el que usted quiere leer de línea en línea, usted querra usar una de las funciones `getline()`. Versiones sobrecargadas de `get()`

Versiones sobrecargadas de **get()**

La función get() también viene en tres versiones sobrecargadas: una sin argumentos que retorna el siguiente carácter usando un valor de retorno int; una que recoge un carácter dentro de su argumento char usando una referencia; y una que almacena directamente dentro del buffer subyacente de otro objeto iostream. Este último se explora después en el capítulo.

Leyendo bytes sin formato

Si usted sabe exactamente con que esta tratando y quiere mover los bytes directamente dentro de una variable, un array, o una estructura de memoria, puede usar la función de E/S sin formatear read(). El primer argumento para esta función es un puntero a la destinación en memoria, y el segundo es el número de bytes para leer. Es especialmente útil su usted ha almacenado previamente la información a un fichero, por ejemplo, en formato binario usando la función miembro complementaria write() para el stream de salida (usando el mismo compilador, por supuesto). Verá ejemplos de todas estas funciones más adelante.

5.3. Manejo errores de stream

El extractor de Date mostrado antes activa el bit de error de un stream bajo ciertas condiciones. ¿Como sabe un usuario que este error ha ocurrido? Puede detectar errores del stream llamando a ciertas funciones miembro del stream para ver si tenemos un estado de error, o si a usted no le preocupa qué tipo de error ha pasado, puede evaluar el stream en un contexto Booleano. Ambas técnicas derivan del estado del bit de error de un stream.

5.3.1. Estados del stream

La clase ios_base, desde la que ios deriva,[2] define cuatro banderas que puede usar para comprobar el estado de un stream:

Bandera

Significado

badbit

Algún error fatal (quizás físico) ha ocurrido. El stream debe considerarse no usable.

eofbit

Ha ocurrido un final de entrada (ya sea por haber encontrado un final físico de un stream de fichero o por que el usuario ha terminado el stream de consola, (usando un Ctrl-Z o Ctrl-D).

failbit

Una operación de E/S ha fallado, casi seguro que por datos inválidos (p.e. encontrar letras cuando se intentaba leer un número). El stream todavía se puede usar. El failbit también se activa cuando ocurre un final de entrada.

goodbit

[2] Por esa razón usted puede escribir ios::failbit en lugar de ios_base::failbit para ahorrar pulsaciones.

Todo va bien; no hay errores. La final de la entrada todavía no ha ocurrido.

Puede comprobar si alguna de estas condiciones ha ocurrido llamando a la función miembro correspondiente que retorna un valor Booleano indicando cual de estas ha sido activada. La función miembro de `stream good()` retorna cierto si ninguno de los otros tres bits se han activado. La función `eof()` retorna cierto si `eofbit` está activado, que ocurre con un intento de leer de un `stream` que ya no tiene datos (generalmente un fichero). Como el final de una entrada ocurre en C++ cuando tratamos de leer pasado el final del medio físico, `failbit` también se activa para indicar que los datos esperados no han sido correctamente leídos. La función `fail()` retorna cierto si `failbit` o `badbit` están activados, y `bad()` retorna cierto solo si `badbit` está activado.

Una vez alguno de los bit de error de un `stream` se activa, permanece activo, cosa que no siempre es lo que se quiere. Cuando leemos un fichero, usted puede querer colocarse en una posición anterior en el fichero antes de su final. Simplemenete moviendo el puntero del fichero no se desactiva el `eofbit` o el `failbit`; debe hacerlo usted mismo con la función `clear()`, haciendo algo así:

```
myStream.clear(); // Clears all error bits
```

Después de llamar a `clear()`, `good()` retornará cierto si es llamada inmediatamente. Como vió en el extractor de `Date` antes, la función `setstate()` activa los bits que usted le pasa.¿Eso significa que setstate no afecta a los otros bits? Si ya esta activo, permanece activo. Si usted quiere activar ciertos bits pero en el mismo momento, desactivar el resto, usted puede llamar una versión sobrecargada de `clear()`, pasandole una expresion binaria representando los bits que quiere que se activen, así:

```
myStream.clear(ios::failbit | ios::eofbit);
```

La mayoría del tiempo usted no estará interesado en comprobar los bits de estado del `stream` individualmente. Generalmente usted simplemente quiere conocer si todo va bien. Ese es el caso cuando quiere leer un fichero del principio al final; usted quiere saber simplemente cuando la entrada de datos se ha agotado. Puede usar una conversion de la función definida para void* que es automáticamente llamada cuando un `stream` esta en una expresión booleana. Leer un `stream` hasta el final de la entrada usando este idioma se parece a lo siguiente:

```
int i;
while(myStream >> i)
  cout << i << endl;
```

Recuerde que `operator>>()` retorna su argumento `stream`, así que la sentencia `while` anterior comprueba el `stream` como una expresión booleana. Este ejemplo particular asume que el `stream` de entrada myStream contiene enteros separados por un espacio en blanco. La función `ios_base::operator void*()` simplemente llama a `good()` en su `stream` y retorna el resultado.[3] Como la mayoría de operaciones de `stream` retornan su `stream`, usar ese idioma es conveniente.

[3] Es común el uso de `operator void*()` en vez de `operator bool()` porque las conversiones implícitas de booleano a entero pueden causar sorpresas; pueden emplazarle incorrectamente un stream en un contexto donde una conversion a integer puede ser aplicada. La función `operator void*()` solo será llamada implícitamente en el cuerpo de una expresión booleana.

5.3.2. Streams y excepciones

Los `iostream` han existido como parte de C++ mucho antes que hubieran excepciones, luego comprobar el estado de un `stream` manualmente era la manera en que se hacia. Para mantener la compatibilidad, este es todavía el status quo, pero los modernos `iostream` pueden lanzar excepciones en su lugar. La función miembro de `stream` `exceptions()` toma un parámetro representando los bits de estado para los que usted quiere lanzar la excepcion. Siempre que el stream encuentra este estado,este lanza una excepcion de tipo `std::ios_base::failure`, que hereda de `std::exception`.

Aunque usted puede disparar una excepción para alguno de los cuatro estados de un `stream`, no es necesariamente una buena idea activar las excepciones para cada uno de ellos. Tal como explica el Capítulo uno, se usan las excepciones para condiciones verdaderamente excepcionales, ¡pero el final de un fichero no solo no es excepcional! ¡Es lo que se espera! Por esta razón, solo debe querer activar las excepciones para errores representados por `badbit`, que deberia ser como esto:

```
myStream.exceptions(ios::badbit);
```

Usted activa las excepciones `stream` por `stream`, ya que `exceptions()` es una función miembro para los `streams`. La función `exceptions()` retorna una máscara de bits [4] (de tipo `iostate`, que es un tipo dependiente del compilador convertible a int) indicando que estados de stream causarán excepciones. Si estos estados ya han sido activados, la excepción será lanzada inmediatamente. Por supuesto, si usa excepciones en conexiones a streams, debería estar preparado paracapturarlas, lo que quiere decir que necesita envolver todos los stream bon bloques try que tengan un manejador ios::failure. Muchos programadores encuentran tedioso y simplemente comprueban manualmente donde esperan encontrar errores (ya que, por ejemplo, no esperan encontrar bad() al retornar true la mayoria de veces). Esto es otra razón que tienen los streams para que el lanzamiento de excepciones sea opcional y no por defecto. en cualquier caso, usted peude elegir como quiere manejar los errores de stream. Por las mismas razones que recomendamos el uso de excepciones para el manejo de rrores en otros contextos, lo hacemos aqui.

5.4. Iostreams de fichero

Manipular ficheros con `iostream` es mucho más fácil y seguro que usar `stdio` en C. Todo lo que tiene que hacer es crear un objeto - el constructor hace el trabajo. No necesita cerrar el fichero explícitamente (aunque puede, usando la función miembro `close()`) porque el destructor lo cerrará cuando el objeto salga del ámbito. Para crear un fichero que por defecto sea de entrada, cree un objeto `ifstream` . Para crear un fichero que por defecto es de salida, cree un objeto `ofstream`. Un `fstream` puede hacer ambas cosas.

Las clases de `stream` de fichero encajan dentro de las clases `iostream` como se muestra en la siguiente figura:

Como antes, las clases que usted usa en realidad son especializaciones de plantillas definidas por definiciones de tipo. Por ejemplo, `ifstream`, que procesa ficheros de char, es definida como:

[4] un tipo integral usado para alojar bits aislados.

```
typedef basic_ifstream<char> ifstream;
```

5.4.1. Un ejemplo de procesado de fichero.

Aqui tiene un ejemplo que muestra algunas de las características discutidas antes. Nótese que la inclusión de <fstream> para delarar las clases de fichero de E/S. Aunque en muchas plataformas esto también incluye <iostream> automáticamente, los compiladores no están obligados a hacer esto. Si usted quiere compatibilidad, incluya siempre ambas cabeceras.

```
//: C04:Strfile.cpp
// Stream I/O with files;
// The difference between get() & getline().
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  const int SZ = 100; // Buffer size;
  char buf[SZ];
  {
    ifstream in("Strfile.cpp"); // Read
    assure(in, "Strfile.cpp"); // Verify open
    ofstream out("Strfile.out"); // Write
    assure(out, "Strfile.out");
    int i = 1; // Line counter

    // A less-convenient approach for line input:
    while(in.get(buf, SZ)) { // Leaves \n in input
      in.get(); // Throw away next character (\n)
      cout << buf << endl; // Must add \n
      // File output just like standard I/O:
      out << i++ << ": " << buf << endl;
    }
  } // Destructors close in & out

  ifstream in("Strfile.out");
  assure(in, "Strfile.out");
  // More convenient line input:
  while(in.getline(buf, SZ)) { // Removes \n
    char* cp = buf;
    while(*cp != ':')
      ++cp;
    cp += 2; // Past ": "
    cout << cp << endl; // Must still add \n
  }
} ///:~
```

La creación tanto del ifstream como del ofstream están seguidas de un `assure()` para garantizar que el fichero ha sido abierto exitosamente. El objeto resultante, usado en una situación donde el compilador espera un resultado booleano, produce un valor que indica éxito o fracaso.

El primer `while` demuestra el uso de dos formas de la función `get()`. La primera toma los carácteres dentro de un buffer y pone un delimitador cero en el buffer cuando bien SZ-1 carácteres han sido leidos o bien el tercer argumento (que por defecto es '\n') es encontrado. La función `get()` deja el carácter delimitador en el `stream` de entrada, así que este delimitador debe ser eliminado via `in.get()` usando la forma de `get()` sin argumentos. Puede usar tambien la función miembro `ignore()`, que tiene dos parámetros por defecto. El primer argumento es el número de carácteres para descartar y por defecto es uno. El segundo argumento es el carácter en el que `ignore()` se detiene (después de extraerlo) y por defecto es EOF.

A continuación, se muestran dos sentencias de salida similares: una hacia `cout` y la otra al fichero de salida. Nótese la conveniencia aquí - no necesita preocuparse del tipo de objeto porque las sentencias de formateo trabajan igual con todos los objetos `ostream`. El primero hace eco de la linea en la salida estándar, y el segundo escribe la línea hacia el fichero de salida e incluye el número de línea.

Para demostrar `getline()`, abra el fichero recién creado y quite los números de linea. Para asegurarse que el fichero se cierra correctamente antes de abrirlo para la lectura, usted tiene dos opciones. Puede envolver la primera parte del programa con llaves para forzar que el objeto `out` salga del ámbito, llamando así al destructor y cerrando el fichero, que es lo que se hace aquí. Tambien puede l lamar a `close()` para ambos ficheros; si hace esto, puede despues rehusar el objeto de entrada llamando a la función miembro `open()`.

El segundo `while` muestra como `getline()` borra el caracter terminador (su tercer argumento, que por defecto es '\n') del `stream` de entrada cuando este es encontrado. Aunque `getline()`, como `get()`, pone un cero en el buffer, este todavía no inserta el carácter de terminación.

Este ejemplo, así como la mayoría de ejemplos en este capítulo, asume que cada llamada a alguna sobrecarga de `getline()` encontrará un carácter de nueva línea. Si este no es el caso, la estado `eofbit` del `stream` será activado y la llamada a `getline()` retornará falso, causando que el programa pierda la última línea de la entrada.

5.4.2. Modos de apertura

Puede controlar la manera en que un fichero es abierto sobreescribiendo los argumentos por defecto del constructor. La siguiente tabla muestra las banderas que controlan el modo de un fichero:

Bandera

Función

`ios::in`

Abre el fichero de entrada. Use esto como un modo de apertura para un `ofstream` para prevenir que un fichero existente sea truncado.

`ios::out`

Abre un fichero de salida. Cuando es usado por un `ofstream` sin `ios::app`, `ios::ate` o `ios::in`, `ios::trunc` es implicado.

`ios::app`

Abre un fichero de salida para solo añadir .

`ios::ate`

Abre un fichero existente (ya sea de entrada o salida) y busca el final.

`ios::trunc`

Trunca el fichero antiguo si este ya existe.

`ios::binary`

Abre un fichero en modo binario. Por defecto es en modo texto.

Puede combinar estas banderas usando la operación `or` para bits

El flag binario, aun siendo portable, solo tiene efecto en algunos sistemas no UNIX, como sistemas operativos derivados de MS-DOS, que tiene convenciones especiales para el almacenamiento de delimitadores de final de línea. Por ejemplo, en sistemas MS-DOS en modo texto (el cual es por defecto), cada vez que usted inserta un nuevo carácter de nueva línea ('\n'), el sistema de ficheros en realidad inserta dos carácteres, un par retorno de carro/fin de línea (CRLF), que es el par de carácteres ASCII 0x0D y 0x0A. En sentido opuesto, cuando usted lee este fichero de vuelta a memoria en modo texto, cada ocurrencia de este par de bytes causa que un '\n' sea enviado al programa en su lugar. Si quiere sobrepasar este procesado especial, puede abrir el fichero en modo binario. El modo binario no tiene nada que ver ya que usted puede escribir bytes sin formato en un fichero - siempre puede (llamando a `write()`). Usted debería, por tanto, abrir un fichero en modo binario cuando vaya a usar `read()` o `write()`, porque estas funciones toman un contador de bytes como parámetro. Tener carácteres extra '\r' estropeará su contador de bytes en estas instancias. Usted también puede abrir un fichero en formato binario si va a usar comandos de posicionamiento en el `stream` que se discuten más adelante.

Usted puede abrir un fichero tanto para entrada como salida declarando un objeto `fstream`. cuando declara un objeto `fstream`, debe usar suficientes banderas de modos de apertura mencionados antes para dejar que el sistema de ficheros sepa si quiere leer, escribir, o ambos. Para cambiar de salida a entrada, necesita o bien limpiar el `stream` o bien cambiar la posición en el fichero. Para cambiar de entrada a salida, cambie la posicion en el fichero. Para crear un fichero usando un objeto `fstream`, use la bandera de modo de apertura `ios::trunc` en la llamada al constructor para usar entrada y salida.

5.5. Almacenamiento de `iostream`

Las buenas prácticas de diseño dictan que, cuando cree una nueva clase, debe esforzarse en ocultar los detalles de la implementación subyacente tanto como sea posible al usuario de la clase. Usted le muestra solo aquello que necesita conocer y el resto se hace privado para evitar confusiones. Cuando usamos insertadores y extractores, normalmente usted no conoce o tiene cuidado con los bytes que se consumen o se producen, ya que usted está tratando con E/S estándar, ficheros, memoria, o alguna nueva clase o dispositivo creado.

Llega un momento, no obstante, en el que es importante comunicar con la parte del `iostream` que produce o consume bytes. Para proveer esta parte con una interfaz común y esconder todavía su implementación subyacente, la librería estándar la abstrae dentro de su clase, llamada `streambuf`. Cada objeto `iostream` contiene un puntero a alguna clase de `streambuf`. (El tipo depende de que se esté tratando con E/S estándar, ficheros, memoria, etc.). Puede acceder al `streambuf` directamente; por ejemplo, puede mover bytes sin formatear dentro y fuera del `streambuf` sin formatearlos a través de la encapsulación del `iostream`. Esto es posible llamando a las funciones miembro del objeto `streambuf`.

Actualmente, la cosa más importante que debe conocer es que cada objeto iostream contiene un puntero a un objeto streambuf, y el objeto streambuf tiene algunas funciones miembro que puede llamar si es necesario. Para ficheros y streams de string, hay tipos especializados de buffers de stream, como ilustra la figura siguiente:

Para permitirle el acceso al streambuf, cada objeto iostream tiene una función miembro llamada rdbuf() que retorna el puntero a un objeto streambuf. De esta manera usted puede llamar cualquier función miembro del streambuf subyacente. No obstante, una de las cosas más interesantes que usted puede hacer con el puntero al streambuf es conectarlo con otro objeto iostream usando el operador <<. Esto inserta todos los carácteres del objeto dentro del que está al lado izquierdo del <<. Si quiere mover todos los carácteres de un iostream a otro, no necesita ponerse con el tedioso (y potencialmente inclinado a errores de código) proceso de leer de carácter por carácter o línea por línea. Este es un acercamiento mucho más elegante.

Aqui está un programa muy simple que abre un fichero y manda el contenido a la salida estándar (similar al ejemplo previo):

```
//: C04:Stype.cpp
// Type a file to standard output.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Stype.cpp");
  assure(in, "Stype.cpp");
  cout << in.rdbuf(); // Outputs entire file
} ///:~
```

Un ifstream se crea usando el fichero de código fuente para este programa como argumento. La función assure() reporta un fallo si el fichero no puede ser abierto. Todo el trabajo pasa realmente en la sentencia

```
cout << in.rdbuf();
```

que manda todo el contenido del fichero a cout. No solo es un código más sucinto, a menudo es más eficiente que mover los byte de uno en uno.

Una forma de get() escribe directamente dentro del streambuf de otro objeto. El primer argumento es una referencia al streambuf de destino, y el segundo es el carácter de terminación ('\n' por defecto), que detiene la función get(). Así que existe todavía otra manera de imprimir el resultado de un fichero en la salida estándar:

```
//: C04:Sbufget.cpp
// Copies a file to standard output.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
```

```
  ifstream in("Sbufget.cpp");
  assure(in);
  streambuf& sb = *cout.rdbuf();
  while(!in.get(sb).eof()) {
    if(in.fail())          // Found blank line
      in.clear();
    cout << char(in.get()); // Process '\n'
  }
} ///:~
```

La función `rdbuf()` retorna un puntero, que tiene que ser desreferenciado para satisfacer las necesidades de la función para ver el objeto. Los buffers de `stream` no estan pensados para ser copiados (no tienen contructor de copia), por lo que definimos `sb` como una referencia al buffer de `stream` de `cout`. Necesitamos las llamadas a `fail()` y `clear()` en caso de que el fichero de entrada tenga una línea en blanco (este la tiene). Cuando esta particular versión sobrecargada de `get()` vee dos carácteres de nueva línea en una fila (una evidencia de una línea en blanco), activa el bit de error del `stream` de entrada, asi que se debe llamar a `clear()` para resetearlo y que así el `stream` pueda continuar siendo leído. La segunda llamada a `get()` extrae y hace eco de cualquier delimitador de nueva línea. (Recuerde, la función `get()` no extrae este delimitador como sí lo hace `getline()`).

Probablemente no necesitará usar una técnica como esta a menudo, pero es bueno saber que existe.[5]

5.6. Buscar en iostreams

Cada tipo de iostream tiene el concepto de donde está el 'siguiente' carácter que proviene de (si es un `istream`) o que va hacia (si es un `ostream`). En algunas situaciones, puede querer mover la posición en este `stream`. Puede hacer esto usando dos modelos: uno usa una localización absoluta en el stream llamada `streampos`; el segundo trabaja como las funciones `fseek()` de la librería estándar de C para un fichero y se mueve un número dado de bytes desde el principio, final o la posición actual en el fichero.

El acercamiento de `streampos` requiere que primero llame una función 'tell':(`tellp()` para un `ostream` o `tellg()` para un `istream`. (La 'p' se refiere a 'put pointer' y la 'g' se refiere a 'get pointer'). Esta función retorna un `streampos` que puede usar después en llamadas a `seekp()` para un `ostream` o `seekg()` para un `ostream` cuando usted quiere retornar a la posición en el `stream`.

La segunda aproximación es una búsqueda relativa y usa versiones sobrecargadas de `seekp()` y `seekg()`. El primer argumento es el número de carácteres a mover: puede ser positivo o negativo. El segundo argumento es la dirección desde donde buscar:

`ios::beg`

Desde el principio del `stream`

`ios::cur`

Posición actual del `stream`

[5] Un tratado mucho más en profundidad de buffers de stream y streams en general puede ser encontrado en[?, ?, ?, ?]

`ios::end`

Desde el principio del `stream`

Aquí un ejemplo que muestra el movimiento por un fichero, pero recuerde, no esta limitado a buscar en ficheros como lo está con `stdio` de C. Con C++, puede buscar en cualquier tipo de `iostream` (aunque los objetos `stream` estándar, como `cin` y `cout`, lo impiden explícitamente):

```
//: C04:Seeking.cpp
// Seeking in iostreams.
#include <cassert>
#include <cstddef>
#include <cstring>
#include <fstream>
#include "../require.h"
using namespace std;

int main() {
  const int STR_NUM = 5, STR_LEN = 30;
  char origData[STR_NUM][STR_LEN] = {
    "Hickory dickory dus. . .",
    "Are you tired of C++?",
    "Well, if you have,",
    "That's just too bad,",
    "There's plenty more for us!"
  };
  char readData[STR_NUM][STR_LEN] = {{ 0 }};
  ofstream out("Poem.bin", ios::out | ios::binary);
  assure(out, "Poem.bin");
  for(int i = 0; i < STR_NUM; i++)
    out.write(origData[i], STR_LEN);
  out.close();
  ifstream in("Poem.bin", ios::in | ios::binary);
  assure(in, "Poem.bin");
  in.read(readData[0], STR_LEN);
  assert(strcmp(readData[0], "Hickory dickory dus. . .")
    == 0);
  // Seek -STR_LEN bytes from the end of file
  in.seekg(-STR_LEN, ios::end);
  in.read(readData[1], STR_LEN);
  assert(strcmp(readData[1], "There's plenty more for us!")
    == 0);
  // Absolute seek (like using operator[] with a file)
  in.seekg(3 * STR_LEN);
  in.read(readData[2], STR_LEN);
  assert(strcmp(readData[2], "That's just too bad,") == 0);
  // Seek backwards from current position
  in.seekg(-STR_LEN * 2, ios::cur);
  in.read(readData[3], STR_LEN);
  assert(strcmp(readData[3], "Well, if you have,") == 0);
  // Seek from the begining of the file
  in.seekg(1 * STR_LEN, ios::beg);
  in.read(readData[4], STR_LEN);
  assert(strcmp(readData[4], "Are you tired of C++?")
    == 0);
} ///:~
```

Este programa escribe un poema a un fichero usando un `stream` de salida binaria. Como reabrimos como un `ifstream`, usamos `seekg()` para posicionar el 'get pointer'. Como puede ver, puede buscar desde el principio o el final del archivo o desde la posición actual del archivo. Obviamente, debe proveer un número positivo para mover desde el principio del archivo y un número negativo para mover hacia atrás.

Ahora que ya conoce el `streambuf` y como buscar, ya puede entender un método alternativo (aparte de usar un objeto `fstream`) para crear un objeto `stream` que podrá leer y escribir en un archivo. El siguiente código crea un `ifstream` con banderas que dicen que es un fichero de entrada y de salida. Usted no puede escribir en un `ifstream`, así que necesita crear un `ostream` con el buffer subyacente del `stream`:

```
ifstream in("filename", ios::in | ios::out);
ostream out(in.rdbuf());
```

Debe estar preguntándose que ocurre cuando usted lee en uno de estos objetos. Aqui tiene un ejemplo:

```
//: C04:Iofile.cpp
// Reading & writing one file.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Iofile.cpp");
  assure(in, "Iofile.cpp");
  ofstream out("Iofile.out");
  assure(out, "Iofile.out");
  out << in.rdbuf(); // Copy file
  in.close();
  out.close();
  // Open for reading and writing:
  ifstream in2("Iofile.out", ios::in | ios::out);
  assure(in2, "Iofile.out");
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Print whole file
  out2 << "Where does this end up?";
  out2.seekp(0, ios::beg);
  out2 << "And what about this?";
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
} ///:~
```

Las primeras cinco líneas copian el código fuente de este programa en un fichero llamado iofile.out y después cierra los ficheros. Esto le da un texto seguro con el que practicar. Entonces, la técnica antes mencionada se usa para crear dos objetos que leen y escriben en el mismo fichero. En `cout << in2.rebuf()`, puede ver como puntero 'get' es inicializado al principio del fichero. El puntero 'put', en cambio, se coloca en el final del fichero para que 'Where does this end up' aparezca añadido al fichero. No obstante, si el puntero 'put' es movido al principio con un `seekp(-`

), todo el texto insertado sobreescribe el existente. Ambas escrituras pueden verse cuando el puntero 'get' se mueve otra vez al principio con seekg(), y el fichero se muestra. El fichero es automáticamente guardado cuando out2 sale del ámbito y su destructor es invocado.

5.7. Iostreams de string

Un stream de cadena funciona directamente en memoria en vez de con ficheros o la salida estándar. Usa las mismas funciones de lectura y formateo que usó con cin y cout para manipular bits en memoria. En ordenadores antiguos, la memoria se refería al núcleo, con lo que este tipo de funcionalidad se llama a menudo formateo en el núcleo.

Los nombres de clases para streams de cadena son una copia de los streams de ficheros. Si usted quiere crear un stream de cadena para extraer carácteres de él, puede crear un istringstream. Si quiere poner carácteres en un stream de cadena, puede crear un ostringstream. Todas las declaraciones para streams de cadena están en la cabecera estándar <sstream>. Como es habitual, hay plantillas de clases dentro de la jerarquia de los iostreams, como se muestra en la siguiente figura:

5.7.1. Streams de cadena de entrada

Para leer de un string usando operaciones de stream, cree un objeto istringstream inicializado con el string. El siguiente programa muestra como usar un objeto istringstream:

```
//: C04:Istring.cpp
// Input string streams.
#include <cassert>
#include <cmath>  // For fabs()
#include <iostream>
#include <limits> // For epsilon()
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream s("47 1.414 This is a test");
  int i;
  double f;
  s >> i >> f; // Whitespace-delimited input
  assert(i == 47);
  double relerr = (fabs(f) - 1.414) / 1.414;
  assert(relerr <= numeric_limits<double>::epsilon());
  string buf2;
  s >> buf2;
  assert(buf2 == "This");
  cout << s.rdbuf(); // " is a test"
} ///:~
```

Puede ver que es un acercamiento más flexible y general para transformar cadenas de carácteres para valores con tipo que la librería de funciones del estándar

de C, como atof() o atoi(), aunque esta última puede ser más eficaz para las conversiones individuales.

En la expresión s >> i >> f, el primer número se extrae en i, y en el segundo en f. Este no es 'el primer conjunto de carácteres delimitado por espacios en blanco' por que depende del tipo de datos que está siendo extraído. Por ejemplo, si la cadena fuera '1.414 47 This is a test', entonces i tomaría el valor 1 porque la rutina de entrada se pararía en el punto decimal. Entonces f tomaría 0.414. Esto puede ser muy útil i si quiere partir un número de coma flotante entre la parte entera y la decimal. De otra manera parecería un error. El segundo assert() calcula el error relativo entre lo que leemos y lo que esperamos; siempre es mejor hacer esto que comparar la igualdad de números de coma flotante. La constante devuelta por epsilon(), definida en <limits>, representa la epsilon de la máquina para números de doble precisión, el cual es la mejor tolerancia que se puede esperar para satisfacer las comparaciones de double.[6].

Como debe haber supuesto, buf2 no toma el resto del string, simplemente la siguiente palabra delimitada por espacios en blanco. En general, el mejor usar el extractor en iostreams cuando usted conoce exactamente la secuencia de datos en el stream de entrada y los convierte a algún otro tipo que un string de carácteres. No obstante, si quiere extraer el resto del string de una sola vez y enviarlo a otro iostream, puede usar rdbuf() como se muestra.

Para probar el extractor de Date al principio de este capítulo, hemos usado un stream de cadena de entrada con el siguiente programa de prueba:

```
//: C04:DateIOTest.cpp
//{L} ../C02/Date
#include <iostream>
#include <sstream>
#include "../C02/Date.h"
using namespace std;

void testDate(const string& s) {
  istringstream os(s);
  Date d;
  os >> d;
  if(os)
    cout << d << endl;
  else
    cout << "input error with \"" << s << "\"" << endl;
}

int main() {
  testDate("08-10-2003");
  testDate("8-10-2003");
  testDate("08 - 10 - 2003");
  testDate("A-10-2003");
  testDate("08 %10/2003");
} ///:~
```

Cada literal de cadena en main() se pasa por referencia a testDate(), que a su vez lo envuelve en un istringstream con lo que podemos probar el extractor

[6] Para más información sobre la epsilon de la máquina y el cómputo de punto flotante en general, vea el artículo de Chuck, "The Standard C Library, Part 3", C/C++ Users Journal, Marzo 1995, disponible en www.freshsources.com/1995006a.htm

de stream que escribimos para los objetos Date. La función testDate() también empieza por probar el insertador, operator<<().

5.7.2. Streams de cadena de salida

Para crear un stream de cadena de salida, simplemente cree un objeto ostringstream, que maneja un buffer de carácteres dinamicamente dimensionado para guardar cualquier cosas que usted inserte. Para tomar el resultado formateado como un objeto de string, llame a la función miembro setr(). Aqui tiene un ejemplo:

```
//: C04:Ostring.cpp {RunByHand}
// Illustrates ostringstream.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
  cout << "type an int, a float and a string: ";
  int i;
  float f;
  cin >> i >> f;
  cin >> ws; // Throw away white space
  string stuff;
  getline(cin, stuff); // Get rest of the line
  ostringstream os;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << "string = " << stuff << endl;
  string result = os.str();
  cout << result << endl;
} ///:~
```

Esto es similar al ejemplo Istring.cpp anterior que pedía un int y un float. A continueación una simple ejecución (la entrada por teclado está escrita en negrita).

```
type an int, a float and a string: FIXME:10 20.5 the end
integer = 10
float = 20.5
string = the end
```

Puede ver que, como otros stream de salida, puede usar las herramientas ordinarias de formateo, como el operador << y endl, para enviar bytes hacia el ostringstream. La función str() devuelve un nuevo objeto string cada vez que usted la llama con lo que el stringbuf contenido permanece inalterado.

En el capítulo previo, presentamos un programa, HTMLStripper.cpp, que borraba todas las etiqietas HTML y los códigos especiales de un fichero de texto. Como prometíamos, aqui está una versión más elegante usando streams de cadena.

```
//: C04:HTMLStripper2.cpp {RunByHand}
//{L} ../C03/ReplaceAll
// Filter to remove html tags and markers.
#include <cstddef>
#include <cstdlib>
```

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "../C03/ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) throw(runtime_error) {
  size_t leftPos;
  while((leftPos = s.find('<')) != string::npos) {
    size_t rightPos = s.find('>', leftPos+1);
    if(rightPos == string::npos) {
      ostringstream msg;
      msg << "Incomplete HTML tag starting in position "
          << leftPos;
      throw runtime_error(msg.str());
    }
    s.erase(leftPos, rightPos - leftPos + 1);
  }
  // Remove all special HTML characters
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  replaceAll(s, " ", " ");
  // Etc...
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper2 InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  // Read entire file into string; then strip
  ostringstream ss;
  ss << in.rdbuf();
  try {
    string s = ss.str();
    cout << stripHTMLTags(s) << endl;
    return EXIT_SUCCESS;
  } catch(runtime_error& x) {
    cout << x.what() << endl;
    return EXIT_FAILURE;
  }
} ///:~
```

En este programa leemos el fichero entero dentro de un `string` insertando una llamada `rdbuf()` del `stream` de fichero al `ostringstream`. Ahora es fácil buscar parejas de delimitadores HTML y borrarlas sin tener que preocuparnos de límites de líneas como teniamos con la versión previa en el Capítulo 3.

El siguiente ejemplo muestra como usar un stream de cadena bidireccional (esto es, lectura/escritura):

```
//: C04:StringSeeking.cpp {-bor}{-dmc}
```

```
// Reads and writes a string stream.
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
  string text = "We will hook no fish";
  stringstream ss(text);
  ss.seekp(0, ios::end);
  ss << " before its time.";
  assert(ss.str() ==
    "We will hook no fish before its time.");
  // Change "hook" to "ship"
  ss.seekg(8, ios::beg);
  string word;
  ss >> word;
  assert(word == "hook");
  ss.seekp(8, ios::beg);
  ss << "ship";
  // Change "fish" to "code"
  ss.seekg(16, ios::beg);
  ss >> word;
  assert(word == "fish");
  ss.seekp(16, ios::beg);
  ss << "code";
  assert(ss.str() ==
    "We will ship no code before its time.");
  ss.str("A horse of a different color.");
  assert(ss.str() == "A horse of a different color.");
} ///:~
```

Como siempre para mover el puntero de inserción, usted llama a `seekp()`, y para reposicionar el fichero de lectura, usted llama a `seekg()`. Incluso aunque no lo hemos mostrado con este ejemplo, los `stream` de cadeana son un poco más permisivos que los `stream` de fichero ya que podemos cambiar de lectura a escritura y viceversa en cualquier momento. No necesita reposicionar el puntero de lectura o de escritura o vaciar el `stream`. Este progrma también ilustra la sobrecarga de `str()` que reemplaza el `stringbuf` contenido en el `stream` con una nueva cadena.

5.8. Formateo de stream de salida

El objetivo del diseño de los `iostream` es permitir que usted pueda mover y/o formatear carácteres fácilmente. Ciertamente no podria ser de mucha utilidad si no se pudiera hacer la mayoria de los formateos provistos por la familia de funciones de `printf()` en C. Es esta sección, usted aprenderá todo sobre las funciones de formateo de salida que estan disponibles para `iostream`, con lo que puede formatear los bytes de la manera que usted quiera.

Las funciones de formateo en iostream pueden ser algo cunfusas al principio porque a menudo existe más de una manera de controlar el formateo: a través de funciones miembro y manipuladores. Para confundir más las cosas, una función miembro genérica pone banderas de estado para controlar el formateo, como la justificación a la derecha o izquierda, el uso de letras mayúsculas para la notación hexadecimal,

para siempre usar un punto decimal para valores de coma flotante, y cosas así. En el otro lado, funciones miembro separadas activan y leen valores para el caracter de relleno, la anchura del campo, y la precisión.

En un intento de clarificar todo esto, primero examinaremos el formateo interno de los datos de un iostream, y las funciones miembro que pueden modificar estos datos. (Todo puede ser controlado por funciones miembro si se desea). Cubriremos los manipuladores aparte.

5.8.1. Banderas de formateo

La clase `ios` contiene los miembros de datos para guardar toda la información de formateo perteneciente a un `stream`. Algunos de estos datos tiene un rango de valores de datos y son guardados en variables: la precisión de la coma flotante, la anchura del campo de salida, y el carácter usado para rellenar la salida (normalmente un espacio). El resto del formateo es determinado por banderas, que generalmente están combinadas para ahorrar espacio y se llaman colectivamente banderas de formateo. Puede recuperar los valores de las banderas de formateo con la función miembro `ios::flag()`, que no toma argumentos y retorna un objeto de tipo `fmtflags` (usualmente un sinónimo de long) que contiene las banderas de formateo actuales. El resto de funciones hacen cambios en las banderas de formateo y retornan el valor previo de las banderas de formateo.

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

La primera función fuerza que todas las banderas cambien, que a veces es lo que usted quiere. Más a menudo, usted cambia una bandera cada vez usando las otras tres funciones.

El uso de `setf()` puede parecer algo confusa. Para conocer qué versión sobrecargada usar, debe conocer el tipo de la bandera que está cambiando. Existen dos tipos de banderas: las que simplemente estan activadas o no, y aquellas que trabajan en grupo con otras banderas. La banderas que estan encendidas/apagadas son las más simples de entender por que usted las enciende con `setf(fmtflags)` y las apaga con `unsetf(fmtflags)`. Estas banderas se muestran en la siguiente tabla:

bandera activa/inactiva

Efecto

`ios::skipws`

Se salta los espacios en blanco. (Para la entrada esto es por defecto).

`ios::showbase`

Indica la base numérica (que puede ser, por ejemplo, decimal, octal o hexadecimal) cuando imprimimos el valor entero. Los `stream` de entrada tambié reconocen el prefijo de base cuando `showbase` está activo.

`ios::showpoint`

Muestra el punto decimal insertando ceros para valores de coma flotante.

`ios::uppercase`

Muestra A-F mayúsculas para valores hexadecimales y E para científicos.

ios::showpos

Muestra el signo de sumar (+) para los valores positivos

ios::unitbuf

'Unit buffering.' El stream es borrado después de cada inseción.

Por ejemplo, para mostrar el signo de sumar para cout, puede usar cout.setf(ios::showpos). Para dejar de mostrar el signo de sumar, escriba cout.unsetf(ios::showpos).

La bandera de unitbuf controla el almacenamiento unitario, que significa que cada inserción es lanzada a su stream de salida inmediatamente. Esto es útil para hacer recuento de errores, ya que en caso de fallo del programa, sus datos son todavía escritos al fichero de log. El siguiente programa ilustra el almacenamiento unitario.

```
//: C04:Unitbuf.cpp {RunByHand}
#include <cstdlib>  // For abort()
#include <fstream>
using namespace std;

int main() {
  ofstream out("log.txt");
  out.setf(ios::unitbuf);
  out << "one" << endl;
  out << "two" << endl;
  abort();
} ///:~
```

Es necesario activar el almacenamiento unitario antes de que cualquier inserción sea hecha en el stream. Cuando hemos descomentado la llamada a setf(), un compilador en particular ha escrito solo la letra 'o' en el fichero log.txt. Con el almacenamiento unitario, ningun dato se perdió.

El stream de salida estándar cerr tiene el almacenamiento unitario activado por defecto. Hay un coste para el almacenamiento unitario, asi que si un stream de salida se usa intensivamente, no active el almacenamiento unitario a menos que la eficiencia no sea una consideración.

5.8.2. Campos de formateo

El segundo tipo de banderas de formateo trabajan en grupo. Solo una de estas banderas pueden ser activadas cada vez, como los botones de una vieja radio de coche - usted apretaba una y el resto saltaban. Desafortunadamente esto no pasa automáticamente, y usted tiene que poner atención a que bandera está activando para no llamar accidentalmente a la función setf() incorrecta. Por ejemplo, hay una bandera para cada una de las bases numéricas: hexadecimal, decimal y octal. A estas banderas se refiere en conjunto ios::basefield. Si la bandera ios::dec está activa y usted llama setf(ios::hex), usted activará la bandera de ios::hex, pero no desactivará la bandera de ios::dec, resultando en un comportamiento indeterminado. En vez de esto,llame a la segunda forma de la función setf() como esta: setf(ios::hex, ios::basefield) . Esta función primero limpia todos los bits de ios::basefield y luego activa ios::hex. Así, esta forma de setf() asegura que las otras banderas en el grupo 'saltan' cuando usted activa una. El manipulador ios::hex lo hace todo por usted, automáticamente, asi que no tiene que preocupar-

se con los detalles de la implementación interna de esta clase o tener cuidado de que esto es una seria de banderas binarias. Más adelante verá que hay manipuladores para proveer de la funcionalidad equivalente en todas las parts donde usted fuera a usar `setf()`.

Aquí están los grupos de banderas y sus efectos:

`ios::basefield`

Efecto

`ios::dec`

Formatea valores enteros en base 10 (decimal)(Formateo por defecto - ningún prefijo es visible).

`ios::hex`

Formatea valores enteros en base 16 (hexadecimal).

`ios::oct`

Formatea valores enteros en base 8 (octal).

`ios::floatfield`

Efecto

`ios::scientific`

Muestra números en coma flotante en formato científico. El campo precisión indica el numero de dígitos después del punto decimal.

`ios::fixed`

Muestra números en coma flotante en formato fijado. El campo precisión indica en número de dígitos después del punto decimal.

'automatic' (Ninguno de los bits está activado).

El campo precisión indica el número total de dígitos significativos.

`ios::adjustfield`

Efecto

`ios::left`

Valores con alineación izquierda; se llena hasta la derecha con el carácter de relleno.

`ios::right`

Valores con alineación derecha; se llena hasta la izquierda con el carácter de relleno. Esta es la alineación por defecto.

`ios::internal`

Añade carácteres de relleno despues de algún signo inicial o indicador de base, pero antes del valor. (En otras palabras, el signo, si está presente, se justifica a la izquierda mientras el número se justifica a la derecha).

5.8.3. Anchura, relleno y precisión

La variables internas que controlan la anchura del campo de salida, el carácter usado para rellenar el campo de salida, y la precisión para escribir números de coma flotante son escritos y leídos por funciones miembro del mismo nombre.

Función

Efecto

`int ios::width( )`

Retorna la anchura actual. Por defecto es 0. Se usa para la inserción y la extracción.

`int ios::width(int n)`

Pone la anchura, retorna la anchura previa.

`int ios::fill( )`

Retorna el carácter de relleno actual. Por defecto es el espacio.

`int ios::fill(int n)`

Poner el carácter de relleno, retorna el carácter de relleno anterior.

`int ios::precision( )`

Retorna la precisión actual de la coma flotante. Por defecto es 6.

`int ios::precision(int n)`

Pone la precisión de la coma flotante, retorna la precisión anteriot. Vea la tabla `ios::floatfield` para el significado de 'precisión'.

El relleno y la precisión son bastante claras, pero la anchura requiera alguna explicación. Cuando la anchura es 0, insertar un valor produce el mínimo número de carácteres necesario para representar este valor. Una anchura positiva significa que insertar un valor producirá al menos tantos carácteres como la anchura; si el valor tiene menos carácteres que la anchura, el carácter de relleno llena el campo. No obstante, el valor nunca será truncado, con lo que si usted intenta escribir 123 con una anchura de dos, seguirá obteniendo 123. El campo anchura especifica un minimo número de carácteres; no hay forma de especificar un número mínimo.

La anchura también es diferente por que vuelve a ser cero por cada insertador o extractor que puede ser influenciado por este valor. Realmente no es una variable de estado, sino más bien un argumento implícito para los extractores y insertadores. Si quiere una anchura constante, llame a `width()` despues de cada inserción o extracción.

5.8.4. Un ejemplo exhaustivo

Para estar seguros de que usted conoce como llamar a todas las funciones discutidas previamente, aquí tiene un ejemplo que las llama a todas:

```
//: C04:Format.cpp
// Formatting Functions.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;
#define D(A) T << #A << endl; A

int main() {
  ofstream T("format.out");
  assure(T);
  D(int i = 47;)
```

```
  D(float f = 2300114.414159;)
  const char* s = "Is there any more?";

  D(T.setf(ios::unitbuf);)
  D(T.setf(ios::showbase);)
  D(T.setf(ios::uppercase | ios::showpos);)
  D(T << i << endl;) // Default is dec
  D(T.setf(ios::hex, ios::basefield);)
  D(T << i << endl;)
  D(T.setf(ios::oct, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::showbase);)
  D(T.setf(ios::dec, ios::basefield);)
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.fill('0');)
  D(T << "fill char: " << T.fill() << endl;)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::right, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::internal, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T << i << endl;) // Without width(10)

  D(T.unsetf(ios::showpos);)
  D(T.setf(ios::showpoint);)
  D(T << "prec = " << T.precision() << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.unsetf(ios::uppercase);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.precision(20);)
  D(T << "prec = " << T.precision() << endl;)
  D(T << endl << f << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)

  D(T.width(10);)
  T << s << endl;
  D(T.width(40);)
  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)
  T << s << endl;
} ///:~
```

Este ejemplo usa un truco para crear un fichero de traza para que pueda monitorizar lo que está pasando. La macro D(a) usa el preprocesador 'convirtiendo a string' para convertir a en una cadena para mostrar. Entonces se reitera a con lo que la sentencia se ejecuta. La macro envia toda la información a un fichero llamado T, que es

el fichero de traza. La salida es:

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::showbase);
T.setf(ios::uppercase | ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
0X2F
T.setf(ios::oct, ios::basefield);
T << i << endl;
057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300114E+06
T.unsetf(ios::uppercase);
T << endl << f << endl;

2.300114e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.50000000000000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.30011450000000000000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000000000000000
T.width(10);
Is there any more?
T.width(40);
0000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?0000000000000000000000
```

Estudiar esta salida debería clarificar su entendimiento del formateo de las funciones miembro de iostream .

5.9. Manipuladores

Como puede ver en el programa previo, llamar a funciones miembro para operaciones de formateo de stream puede ser un poco tedioso. Para hacer las cosas mas fáciles de leer y escribir, existe un conjunto de manipuladores para duplicar las acciones previstas por las funciones miembro. Las manipuladores son convenientes por que usted puede insertarlos para que actuen dentro de una expresion contenedora; no necesita crear una sentencia de llamada a función separada.

Los manipuladores cambian el estado de un stream en vez de (o además de) procesar los datos. Cuando insertamos un endl en una expresión de salida, por ejemplo, no solo inserta un carácter de nueva linea, sino que ademas termina el stream (esto es, saca todos los carácteres pendientes que han sido almacenadas en el buffer interno del stream pero todavia no en la salida). Puede terminar el stream simplemente asi:

```
cout << flush;
```

Lo que causa una llamada a la función miembro flush(), como esta:

```
cout.flush();
```

como efecto lateral (nada es insertado dentro de stream). Adicionalmente los manipuladores básicos cambirán la base del número a oct (octal), dec (decimal) o hex (hexadecimal).

```
cout << hex << "0x" << i << endl;
```

En este caso, la salida numérica continuará en modo hexadecimal hasta que usted lo cambie insertando o dec o oct en el stream de salida.

También existe un manipulador para la extracción que se 'come' los espacios en blanco:

```
cin >> ws;
```

Los manipuladores sin argumentos son provistos en <iostream>. Esto incluye dec, oct,y hex, que hacen las mismas acciones que, respectivamente, setf(ios::dec, ios::basefield), setf(ios::oct, ios::basefield), y setf(ios::hex, ios::basefield), aunque más sucintamente. La cabecera <iostream> también incluye ws, endl, y flush y el conjunto adicional mostrado aquí:

Manipulador

Efecto

showbase noshowbase

Indica la base numérica (dec, oct, o hex) cuando imprimimos un entero.

showpos noshowpos

Muestra el signo más (+) para valores positivos.

uppercase nouppercase

Muestra mayúsculas A-F para valores hexadecimales, y muestra E para valores

científicos.

showpoint noshowpoint

Muestra punto decimal y ceros arrastrados para valores de coma flotante.

skipws noskipws

Escapa los espacios en blanco en la entrada.

left right internal

Alineación izquierda, relleno a la derecha. Alineación derecha, relleno a la izquierda. Rellenar entre el signo o el indicador de base y el valor.

scientific fixed

Indica la preferencia al mostrar la salida para coma flotante (notación científica versus coma flotante decimal).

5.9.1. Manipuladores con argumentos

Existen seis manipuladores estándar, como `setw()`, que toman argumentos. Están definidos en el fichero de cabecera <iomanip>, y están enumerados en la siguiente tabla:

Manipulador

Efecto

`setiosflags(fmtflags n)`

Equivalente a una llamada a `setf(n)`. La activación continua hasta el siguiente cambio, como `ios::setf()`.

`resetiosflags(fmtflags n)`

Limpia solo las banderas de formato especificadas por n. La activación permanece hasta el siguiente cambio, como `ios::unsetf()`.

`setbase(base n)`

Cambia la base a n, donde n es 10, 8 o 16. (Caulquier otra opción resulta en 0). Si n es cero, la salida es base 10, pero la entrada usa convenciones de C: 10 es 10, 010 es 8, y 0xf es 15. Puede usar también dec, oct y hex para la salida.

`setfill(char n)`

Cambia el carácter de relleno a n, como `ios::fill()`.

`setprecision(int n)`

Cambia la precision a n, como `ios::precision()`.

`setw(int n)`

Cambia la anchura del campo a n, como en `ios::width()`

Si está usando mucho el formateo, usted puede ver como usar los manipuladores en vez de llamar a funciones miembro de stream puede limpiar su código. Como ejemplo, aquí tiene un programa de la sección previa reescrito para usar los manipuladores. (La macro D() ha sido borrada para hacerlo más fácil de leer).

```
//: C04:Manips.cpp
// Format.cpp using manipulators.
#include <fstream>
```

```
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
  ofstream trc("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Is there any more?";

  trc << setiosflags(ios::unitbuf
           | ios::showbase | ios::uppercase
           | ios::showpos);
  trc << i << endl;
  trc << hex << i << endl
      << oct << i << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << resetiosflags(ios::showbase)
      << dec << setfill('0');
  trc << "fill char: " << trc.fill() << endl;
  trc << setw(10) << i << endl;
  trc.setf(ios::right, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc.setf(ios::internal, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc << i << endl; // Without setw(10)

  trc << resetiosflags(ios::showpos)
      << setiosflags(ios::showpoint)
      << "prec = " << trc.precision() << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << resetiosflags(ios::uppercase) << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;
  trc << setprecision(20);
  trc << "prec = " << trc.precision() << endl;
  trc << f << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;

  trc << setw(10) << s << endl;
  trc << setw(40) << s << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << setw(40) << s << endl;
} ///:~
```

Puede ver que un montón de sentencias múltiples han sido condensadas dentro de una sola inserción encadenada. Nótese que la llamada a setiosflags() en que se pasa el OR binario de las banderas. Esto se podría haber hecho también con `setf()` y `unsetf()` como en el ejemplo previo.

```
//: C04:InputWidth.cpp
```

```
// Shows limitations of setw with input.
#include <cassert>
#include <cmath>
#include <iomanip>
#include <limits>
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream is("one 2.34 five");
  string temp;
  is >> setw(2) >> temp;
  assert(temp == "on");
  is >> setw(2) >> temp;
  assert(temp == "e");
  double x;
  is >> setw(2) >> x;
  double relerr = fabs(x - 2.34) / x;
  assert(relerr <= numeric_limits<double>::epsilon());
} ///:~
```

5.9.2.

```
ostream& endl(ostream&);
```

```
cout << "howdy" << endl;
```

```
ostream& ostream::operator<<(ostream& (*pf)(ostream&)) {
   return pf(*this);
}
```

```
//: C04:nl.cpp
// Creating a manipulator.
#include <iostream>
using namespace std;

ostream& nl(ostream& os) {
  return os << '\n';
}

int main() {
  cout << "newlines" << nl << "between" << nl
       << "each" << nl << "word" << nl;
} ///:~
```

```
cout.operator<<(nl) è nl(cout)
```

```
os << '\n';
```

5.9.3.

```
//: C04:Effector.cpp
// Jerry Schwarz's "effectors."
#include <cassert>
#include <limits>  // For max()
#include <sstream>
#include <string>
using namespace std;

// Put out a prefix of a string:
class Fixw {
  string str;
public:
  Fixw(const string& s, int width) : str(s, 0, width) {}
  friend ostream& operator<<(ostream& os, const Fixw& fw) {
    return os << fw.str;
  }
};

// Print a number in binary:
typedef unsigned long ulong;

class Bin {
  ulong n;
public:
  Bin(ulong nn) { n = nn; }
  friend ostream& operator<<(ostream& os, const Bin& b) {
    const ulong ULMAX = numeric_limits<ulong>::max();
    ulong bit = ~(ULMAX >> 1); // Top bit set
    while(bit) {
      os << (b.n & bit ? '1' : '0');
      bit >>= 1;
    }
    return os;
  }
};

int main() {
  string words = "Things that make us happy, make us wise";
  for(int i = words.size(); --i >= 0;) {
    ostringstream s;
    s << Fixw(words, i);
    assert(s.str() == words.substr(0, i));
  }
  ostringstream xs, ys;
  xs << Bin(0xCAFEBABEUL);
  assert(xs.str() ==
    "1100""1010""1111""1110""1011""1010""1011""1110");
  ys << Bin(0x76543210UL);
  assert(ys.str() ==
    "0111""0110""0101""0100""0011""0010""0001""0000");
} ///:~
```

5.10.

5.10.1.

```
//: C04:Cppcheck.cpp
// Configures .h & .cpp files to conform to style
// standard. Tests existing files for conformance.
#include <fstream>
#include <sstream>
#include <string>
#include <cstddef>
#include "../require.h"
using namespace std;

bool startsWith(const string& base, const string& key) {
  return base.compare(0, key.size(), key) == 0;
}

void cppCheck(string fileName) {
  enum bufs { BASE, HEADER, IMPLEMENT, HLINE1, GUARD1,
    GUARD2, GUARD3, CPPLINE1, INCLUDE, BUFNUM };
  string part[BUFNUM];
  part[BASE] = fileName;
  // Find any '.' in the string:
  size_t loc = part[BASE].find('.');
  if(loc != string::npos)
    part[BASE].erase(loc); // Strip extension
  // Force to upper case:
  for(size_t i = 0; i < part[BASE].size(); i++)
    part[BASE][i] = toupper(part[BASE][i]);
  // Create file names and internal lines:
  part[HEADER] = part[BASE] + ".h";
  part[IMPLEMENT] = part[BASE] + ".cpp";
  part[HLINE1] = "//" ": " + part[HEADER];
  part[GUARD1] = "#ifndef " + part[BASE] + "_H";
  part[GUARD2] = "#define " + part[BASE] + "_H";
  part[GUARD3] = "#endif // " + part[BASE] +"_H";
  part[CPPLINE1] = string("//") + ": " + part[IMPLEMENT];
  part[INCLUDE] = "#include \"" + part[HEADER] + "\"";
  // First, try to open existing files:
  ifstream existh(part[HEADER].c_str()),
           existcpp(part[IMPLEMENT].c_str());
  if(!existh) { // Doesn't exist; create it
    ofstream newheader(part[HEADER].c_str());
    assure(newheader, part[HEADER].c_str());
    newheader << part[HLINE1] << endl
              << part[GUARD1] << endl
              << part[GUARD2] << endl << endl
              << part[GUARD3] << endl;
  } else { // Already exists; verify it
    stringstream hfile; // Write & read
    ostringstream newheader; // Write
    hfile << existh.rdbuf();
```

```
    // Check that first three lines conform:
    bool changed = false;
    string s;
    hfile.seekg(0);
    getline(hfile, s);
    bool lineUsed = false;
    // The call to good() is for Microsoft (later too):
    for(int line = HLINE1; hfile.good() && line <= GUARD2;
         ++line) {
      if(startsWith(s, part[line])) {
        newheader << s << endl;
        lineUsed = true;
        if(getline(hfile, s))
          lineUsed = false;
      } else {
        newheader << part[line] << endl;
        changed = true;
        lineUsed = false;
      }
    }
    // Copy rest of file
    if(!lineUsed)
      newheader << s << endl;
    newheader << hfile.rdbuf();
    // Check for GUARD3
    string head = hfile.str();
    if(head.find(part[GUARD3]) == string::npos) {
      newheader << part[GUARD3] << endl;
      changed = true;
    }
    // If there were changes, overwrite file:
    if(changed) {
      existh.close();
      ofstream newH(part[HEADER].c_str());
      assure(newH, part[HEADER].c_str());
      newH << "//@//\n"  // Change marker
           << newheader.str();
    }
  }
  if(!existcpp) { // Create cpp file
    ofstream newcpp(part[IMPLEMENT].c_str());
    assure(newcpp, part[IMPLEMENT].c_str());
    newcpp << part[CPPLINE1] << endl
           << part[INCLUDE] << endl;
  } else { // Already exists; verify it
    stringstream cppfile;
    ostringstream newcpp;
    cppfile << existcpp.rdbuf();
    // Check that first two lines conform:
    bool changed = false;
    string s;
    cppfile.seekg(0);
    getline(cppfile, s);
    bool lineUsed = false;
    for(int line = CPPLINE1;
        cppfile.good() && line <= INCLUDE; ++line) {
      if(startsWith(s, part[line])) {
        newcpp << s << endl;
```

```
          lineUsed = true;
          if(getline(cppfile, s))
            lineUsed = false;
        } else {
          newcpp << part[line] << endl;
          changed = true;
          lineUsed = false;
        }
      }
      // Copy rest of file
      if(!lineUsed)
        newcpp << s << endl;
      newcpp << cppfile.rdbuf();
      // If there were changes, overwrite file:
      if(changed) {
        existcpp.close();
        ofstream newCPP(part[IMPLEMENT].c_str());
        assure(newCPP, part[IMPLEMENT].c_str());
        newCPP << "//@//\n"  // Change marker
               << newcpp.str();
      }
    }
  }

int main(int argc, char* argv[]) {
  if(argc > 1)
    cppCheck(argv[1]);
  else
    cppCheck("cppCheckTest.h");
} ///:~
```

```
// CPPCHECKTEST.h
#ifndef CPPCHECKTEST_H
#define CPPCHECKTEST_H
#endif // CPPCHECKTEST_H
```

```
// PPCHECKTEST.cpp
#include "CPPCHECKTEST.h"
```

5.10.2.

```
//: C04:Showerr.cpp {RunByHand}
// Un-comment error generators.
#include <cstddef>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;
```

```
const string USAGE =
  "usage: showerr filename chapnum\n"
  "where filename is a C++ source file\n"
  "and chapnum is the chapter name it's in.\n"
  "Finds lines commented with //! and removes\n"
  "the comment, appending //(#) where # is unique\n"
  "across all files, so you can determine\n"
  "if your compiler finds the error.\n"
  "showerr /r\n"
  "resets the unique counter.";

class Showerr {
  const int CHAP;
  const string MARKER, FNAME;
  // File containing error number counter:
  const string ERRNUM;
  // File containing error lines:
  const string ERRFILE;
  stringstream edited; // Edited file
  int counter;
public:
  Showerr(const string& f, const string& en,
    const string& ef, int c)
  : CHAP(c), MARKER("//!"), FNAME(f), ERRNUM(en),
    ERRFILE(ef), counter(0) {}
  void replaceErrors() {
    ifstream infile(FNAME.c_str());
    assure(infile, FNAME.c_str());
    ifstream count(ERRNUM.c_str());
    if(count) count >> counter;
    int linecount = 1;
    string buf;
    ofstream errlines(ERRFILE.c_str(), ios::app);
    assure(errlines, ERRFILE.c_str());
    while(getline(infile, buf)) {
      // Find marker at start of line:
      size_t pos = buf.find(MARKER);
      if(pos != string::npos) {
        // Erase marker:
        buf.erase(pos, MARKER.size() + 1);
        // Append counter & error info:
        ostringstream out;
        out << buf << " // (" << ++counter << ") "
            << "Chapter " << CHAP
            << " File: " << FNAME
            << " Line " << linecount << endl;
        edited << out.str();
        errlines << out.str(); // Append error file
      }
      else
        edited << buf << "\n"; // Just copy
      ++linecount;
    }
  }
  void saveFiles() {
    ofstream outfile(FNAME.c_str()); // Overwrites
    assure(outfile, FNAME.c_str());
```

```
    outfile << edited.rdbuf();
    ofstream count(ERRNUM.c_str()); // Overwrites
    assure(count, ERRNUM.c_str());
    count << counter; // Save new counter
  }
};

int main(int argc, char* argv[]) {
  const string ERRCOUNT("../errnum.txt"),
    ERRFILE("../errlines.txt");
  requireMinArgs(argc, 1, USAGE.c_str());
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Allow for other switches:
    switch(argv[1][1]) {
      case 'r': case 'R':
        cout << "reset counter" << endl;
        remove(ERRCOUNT.c_str()); // Delete files
        remove(ERRFILE.c_str());
        return EXIT_SUCCESS;
      default:
        cerr << USAGE << endl;
        return EXIT_FAILURE;
    }
  }
  if(argc == 3) {
    Showerr s(argv[1], ERRCOUNT, ERRFILE, atoi(argv[2]));
    s.replaceErrors();
    s.saveFiles();
  }
} ///:~
```

5.10.3.

```
//: C04:DataLogger.h
// Datalogger record layout.
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iosfwd>
#include <string>
using std::ostream;

struct Coord {
  int deg, min, sec;
  Coord(int d = 0, int m = 0, int s = 0)
  : deg(d), min(m), sec(s) {}
  std::string toString() const;
};

ostream& operator<<(ostream&, const Coord&);

class DataPoint {
  std::time_t timestamp; // Time & day
  Coord latitude, longitude;
  double depth, temperature;
```

```
public:
  DataPoint(std::time_t ts, const Coord& lat,
            const Coord& lon, double dep, double temp)
  : timestamp(ts), latitude(lat), longitude(lon),
      depth(dep), temperature(temp) {}
  DataPoint() : timestamp(0), depth(0), temperature(0) {}
  friend ostream& operator<<(ostream&, const DataPoint&);
};
#endif // DATALOG_H ///:~
```

```
//: C04:DataLogger.cpp {O}
// Datapoint implementations.
#include "DataLogger.h"
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

ostream& operator<<(ostream& os, const Coord& c) {
  return os << c.deg << '*' << c.min << '\''
            << c.sec << '"';
}

string Coord::toString() const {
  ostringstream os;
  os << *this;
  return os.str();
}

ostream& operator<<(ostream& os, const DataPoint& d) {
  os.setf(ios::fixed, ios::floatfield);
  char fillc = os.fill('0'); // Pad on left with '0'
  tm* tdata = localtime(&d.timestamp);
  os << setw(2) << tdata->tm_mon + 1 << '\\'
     << setw(2) << tdata->tm_mday << '\\'
     << setw(2) << tdata->tm_year+1900 << ' '
     << setw(2) << tdata->tm_hour << ':'
     << setw(2) << tdata->tm_min << ':'
     << setw(2) << tdata->tm_sec;
  os.fill(' '); // Pad on left with ' '
  streamsize prec = os.precision(4);
  os << " Lat:"    << setw(9) << d.latitude.toString()
     << ", Long:"  << setw(9) << d.longitude.toString()
     << ", depth:" << setw(9) << d.depth
     << ", temp:"  << setw(9) << d.temperature;
  os.fill(fillc);
  os.precision(prec);
  return os;
} ///:~
```

```
//: C04:Datagen.cpp
// Test data generator.
//{L} DataLogger
```

```
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <fstream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
  time_t timer;
  srand(time(&timer)); // Seed the random number generator
  ofstream data("data.txt");
  assure(data, "data.txt");
  ofstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  for(int i = 0; i < 100; i++, timer += 55) {
    // Zero to 199 meters:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += 1.0 / fraction;
    double newtemp = 150 + rand() % 200; // Kelvin
    fraction = rand() % 100 + 1;
    newtemp += 1.0 / fraction;
    const DataPoint d(timer, Coord(45,20,31),
                      Coord(22,34,18), newdepth,
                      newtemp);
    data << d << endl;
    bindata.write(reinterpret_cast<const char*>(&d),
                  sizeof(d));
  }
} ///:~
```

```
//: C04:Datascan.cpp
//{L} DataLogger
#include <fstream>
#include <iostream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
  ifstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  DataPoint d;
  while(bindata.read(reinterpret_cast<char*>(&d),
         sizeof d))
    cout << d << endl;
} ///:~
```

5.11.

5.11.1.

5.11.2.

```
//: C04:Locale.cpp {-g++}{-bor}{-edg} {RunByHand}
// Illustrates effects of locales.
#include <iostream>
#include <locale>
using namespace std;

int main() {
  locale def;
  cout << def.name() << endl;
  locale current = cout.getloc();
  cout << current.name() << endl;
  float val = 1234.56;
  cout << val << endl;
  // Change to French/France
  cout.imbue(locale("french"));
  current = cout.getloc();
  cout << current.name() << endl;
  cout << val << endl;

  cout << "Enter the literal 7890,12: ";
  cin.imbue(cout.getloc());
  cin >> val;
  cout << val << endl;
  cout.imbue(def);
  cout << val << endl;
} ///:~
```

```
//: C04:Facets.cpp {-bor}{-g++}{-mwcc}{-edg}
#include <iostream>
#include <locale>
#include <string>
using namespace std;

int main() {
  // Change to French/France
  locale loc("french");
  cout.imbue(loc);
  string currency =
    use_facet<moneypunct<char> >(loc).curr_symbol();
  char point =
    use_facet<moneypunct<char> >(loc).decimal_point();
  cout << "I made " << currency << 12.34 << " today!"
       << endl;
} ///:~
```

5.12.

5.13.

```
//: C04:Exercise14.cpp
#include <fstream>
#include <iostream>
#include <sstream>
#include "../require.h"
using namespace std;

#define d(a) cout << #a " ==\t" << a << endl;

void tellPointers(fstream& s) {
  d(s.tellp());
  d(s.tellg());
  cout << endl;
}
void tellPointers(stringstream& s) {
  d(s.tellp());
  d(s.tellg());
  cout << endl;
}
int main() {
  fstream in("Exercise14.cpp");
  assure(in, "Exercise14.cpp");
  in.seekg(10);
  tellPointers(in);
  in.seekp(20);
  tellPointers(in);
  stringstream memStream("Here is a sentence.");
  memStream.seekg(10);
  tellPointers(memStream);
  memStream.seekp(5);
  tellPointers(memStream);
} ///:~
```

```
//: C04:Exercise15.txt
Australia
5E56,7667230284,Langler,Tyson,31.2147,0.00042117361
2B97,7586701,Oneill,Zeke,553.429,0.0074673053156065
4D75,7907252710,Nickerson,Kelly,761.612,0.010276276
9F2,6882945012,Hartenbach,Neil,47.9637,0.0006471644
Austria
480F,7187262472,Oneill,Dee,264.012,0.00356226040013
1B65,4754732628,Haney,Kim,7.33843,0.000099015948475
DA1,1954960784,Pascente,Lester,56.5452,0.0007629529
3F18,1839715659,Elsea,Chelsy,801.901,0.010819887645
Belgium
BDF,5993489554,Oneill,Meredith,283.404,0.0038239127
5AC6,6612945602,Parisienne,Biff,557.74,0.0075254727
6AD,6477082,Pennington,Lizanne,31.0807,0.0004193544
4D0E,7861652688,Sisca,Francis,704.751,0.00950906238
Bahamas
37D8,6837424208,Parisienne,Samson,396.104,0.0053445
```

```
5E98,6384069,Willis,Pam,90.4257,0.00122009564059246
1462,1288616408,Stover,Hazal,583.939,0.007878970561
5FF3,8028775718,Stromstedt,Bunk,39.8712,0.000537974
1095,3737212,Stover,Denny,3.05387,0.000041205248883
7428,2019381883,Parisienne,Shane,363.272,0.00490155
///:~
```

6: Las plantillas en profundidad

6.1.

```
template<class T> class Stack {
  T* data;
  size_t count;
public:
  void push(const T& t);
  // Etc.
};
```

```
Stack<int> myStack; // A Stack of ints
```

6.1.1.

```
template<class T, size_t N> class Stack {
  T data[N];  // Fixed capacity is N
  size_t count;
public:
  void push(const T& t);
  // Etc.
};
```

```
Stack<int, 100> myFixedStack;
```

```
//: C05:Urand.h {-bor}
// Unique randomizer.
#ifndef URAND_H
#define URAND_H
#include <bitset>
#include <cstddef>
#include <cstdlib>
#include <ctime>
using std::size_t;
using std::bitset;

template<size_t UpperBound> class Urand {
  bitset<UpperBound> used;
```

```
public:
  Urand() { srand(time(0)); } // Randomize
  size_t operator()(); // The "generator" function
};

template<size_t UpperBound>
inline size_t Urand<UpperBound>::operator()() {
  if(used.count() == UpperBound)
    used.reset();  // Start over (clear bitset)
  size_t newval;
  while(used[newval = rand() % UpperBound])
    ; // Until unique value is found
  used[newval] = true;
  return newval;
}
#endif // URAND_H ///:~
```

```
//: C05:UrandTest.cpp {-bor}
#include <iostream>
#include "Urand.h"
using namespace std;

int main() {
  Urand<10> u;
  for(int i = 0; i < 20; ++i)
    cout << u() << ' ';
} ///:~
```

6.1.2.

```
template<class T, size_t N = 100> class Stack {
  T data[N];  // Fixed capacity is N
  size_t count;
public:
  void push(const T& t);
  // Etc.
};
```

```
template<class T = int, size_t N = 100>  // Both defaulted
class Stack {
  T data[N];  // Fixed capacity is N
  size_t count;
public:
  void push(const T& t);
  // Etc.
};

Stack<> myStack;  // Same as Stack<int, 100>
```

```
template<class T, class Allocator = allocator<T> >
class vector;
```

```
//: C05:FuncDef.cpp
#include <iostream>
using namespace std;

template<class T> T sum(T* b, T* e, T init = T()) {
  while(b != e)
    init += *b++;
  return init;
}

int main() {
  int a[] = { 1, 2, 3 };
  cout << sum(a, a + sizeof a / sizeof a[0]) << endl; // 6
} ///:~
```

6.1.3.

```
//: C05:TempTemp.cpp
// Illustrates a template template parameter.
#include <cstddef>
#include <iostream>
using namespace std;

template<class T>
class Array { // A simple, expandable sequence
  enum { INIT = 10 };
  T* data;
  size_t capacity;
  size_t count;
public:
  Array() {
    count = 0;
    data = new T[capacity = INIT];
  }
  ~Array() { delete [] data; }
  void push_back(const T& t) {
    if(count == capacity) {
      // Grow underlying array
      size_t newCap = 2 * capacity;
      T* newData = new T[newCap];
      for(size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete [] data;
      data = newData;
      capacity = newCap;
    }
    data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
```

```
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T, template<class> class Seq>
class Container {
  Seq<T> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

```
Seq<T> seq;
```

```
template<class T, template<class> class Seq>
```

```
template<class T, template<class U> class Seq>
```

```
T operator++(int);
```

```
//: C05:TempTemp2.cpp
// A multi-variate template template parameter.
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N> class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
```

```
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T,size_t N,template<class,size_t> class Seq>
class Container {
  Seq<T,N> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  const size_t N = 10;
  Container<int, N, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

```
//: C05:TempTemp3.cpp {-bor}{-msc}
// Template template parameters and default arguments.
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N = 10>  // A default argument
class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T, template<class, size_t = 10> class Seq>
class Container {
  Seq<T> seq;  // Default used
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};
```

```
int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

```
template<class T, template<class, size_t = 10> class Seq>
```

```
//: C05:TempTemp4.cpp {-bor}{-msc}
// Passes standard sequences as template arguments.
#include <iostream>
#include <list>
#include <memory>  // Declares allocator<T>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
         class Seq>
class Container {
  Seq<T> seq; // Default of allocator<T> applied implicitly
public:
  void push_back(const T& t) { seq.push_back(t); }
  typename Seq<T>::iterator begin() { return seq.begin(); }
  typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
  // Use a vector
  Container<int, vector> vContainer;
  vContainer.push_back(1);
  vContainer.push_back(2);
  for(vector<int>::iterator p = vContainer.begin();
      p != vContainer.end(); ++p) {
    cout << *p << endl;
  }
  // Use a list
  Container<int, list> lContainer;
  lContainer.push_back(3);
  lContainer.push_back(4);
  for(list<int>::iterator p2 = lContainer.begin();
      p2 != lContainer.end(); ++p2) {
    cout << *p2 << endl;
  }
} ///:~
```

6.1.4.

```
//: C05:TypenamedID.cpp {-bor}
// Uses 'typename' as a prefix for nested types.

template<class T> class X {
  // Without typename, you should get an error:
  typename T::id i;
public:
  void f() { i.g(); }
};

class Y {
public:
  class id {
  public:
    void g() {}
  };
};

int main() {
  X<Y> xy;
  xy.f();
} ///:~
```

```
//: C05:PrintSeq.cpp {-msc}{-mwcc}
// A print function for Standard C++ sequences.
#include <iostream>
#include <list>
#include <memory>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
         class Seq>
void printSeq(Seq<T>& seq) {
  for(typename Seq<T>::iterator b = seq.begin();
       b != seq.end();)
    cout << *b++ << endl;
}

int main() {
  // Process a vector
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  printSeq(v);
  // Process a list
  list<int> lst;
  lst.push_back(3);
  lst.push_back(4);
  printSeq(lst);
} ///:~
```

```
typename Seq<T>::iterator It;
```

```
typedef typename Seq<It>::iterator It;
```

```
//: C05:UsingTypename.cpp
// Using 'typename' in the template argument list.

template<typename T> class X {};

int main() {
  X<int> x;
} ///:~
```

6.1.5.

```
//: C05:DotTemplate.cpp
// Illustrate the .template construct.
#include <bitset>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

template<class charT, size_t N>
basic_string<charT> bitsetToString(const bitset<N>& bs) {
  return bs. template to_string<charT, char_traits<charT>,
                                allocator<charT> >();
}

int main() {
  bitset<10> bs;
  bs.set(1);
  bs.set(5);
  cout << bs << endl; // 0000100010
  string s = bitsetToString<char>(bs);
  cout << s << endl;  // 0000100010
} ///:~
```

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

```
wstring s = bitsetToString<wchar_t>(bs);
```

6.1.6.

```
template<typename T> class complex {
public:
  template<class X> complex(const complex<X>&);
```

```
complex<float> z(1, 2);
complex<double> w(z);
```

```
template<typename T>
template<typename X>
complex<T>::complex(const complex<X>& c) {/* Body here' */}
```

```
int data[5] = { 1, 2, 3, 4, 5 };
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

```
//: C05:MemberClass.cpp
// A member class template.
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T> class Outer {
public:
  template<class R> class Inner {
  public:
    void f();
  };
};

template<class T> template<class R>
void Outer<T>::Inner<R>::f() {
  cout << "Outer == " << typeid(T).name() << endl;
  cout << "Inner == " << typeid(R).name() << endl;
  cout << "Full Inner == " << typeid(*this).name() << endl;
}

int main() {
  Outer<int>::Inner<bool> inner;
  inner.f();
} ///:~
```

```
Outer == int
Inner == bool
```

```
Full Inner == Outer<int>::Inner<bool>
```

6.2.

```
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
```

```
int z = min<int>(i, j);
```

6.2.1.

```
int z = min(i, j);
```

```
int z = min(x, j); // x is a double
```

```
int z = min<double>(x, j);
```

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
  return (a < b) ? a : b;
}
```

```
//: C05:StringConv.h
// Function templates to convert to and from strings.
#ifndef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>

template<typename T> T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
}

template<typename T> std::string toString(const T& t) {
  std::ostringstream s;
  s << t;
  return s.str();
}
#endif // STRINGCONV_H ///:~
```

```
//: C05:StringConvTest.cpp
#include <complex>
#include <iostream>
#include "StringConv.h"
using namespace std;

int main() {
  int i = 1234;
  cout << "i == \"" << toString(i) << "\"" << endl;
  float x = 567.89;
  cout << "x == \"" << toString(x) << "\"" << endl;
  complex<float> c(1.0, 2.0);
  cout << "c == \"" << toString(c) << "\"" << endl;
  cout << endl;

  i = fromString<int>(string("1234"));
  cout << "i == " << i << endl;
  x = fromString<float>(string("567.89"));
  cout << "x == " << x << endl;
  c = fromString<complex<float> >(string("(1.0,2.0)"));
  cout << "c == " << c << endl;
} ///:~
```

```
i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
c == (1,2)
```

```
//: C05:ImplicitCast.cpp

template<typename R, typename P>
R implicit_cast(const P& p) {
  return p;
}

int main() {
  int i = 1;
  float x = implicit_cast<float>(i);
  int j = implicit_cast<int>(x);
  //! char* p = implicit_cast<char*>(i);
} ///:~
```

```
//: C05:ArraySize.cpp
#include <cstddef>
using std::size_t;

template<size_t R, size_t C, typename T>
void init1(T a[R][C]) {
  for(size_t i = 0; i < R; ++i)
    for(size_t j = 0; j < C; ++j)
```

```
      a[i][j] = T();
}

template<size_t R, size_t C, class T>
void init2(T (&a)[R][C]) {  // Reference parameter
  for(size_t i = 0; i < R; ++i)
    for(size_t j = 0; j < C; ++j)
      a[i][j] = T();
}

int main() {
  int a[10][20];
  init1<10,20>(a);  // Must specify
  init2(a);         // Sizes deduced
} ///:~
```

6.2.2.

```
//: C05:MinTest.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}

const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}

double min(double x, double y) {
  return (x < y) ? x : y;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(1, 2) << endl;      // 1: 1 (template)
  cout << min(1.0, 2.0) << endl;  // 2: 1 (double)
  cout << min(1, 2.0) << endl;    // 3: 1 (double)
  cout << min(s1, s2) << endl;    // 4: knights who (const
                                  //                 char*)
  cout << min<>(s1, s2) << endl;  // 5: say "Ni-!"
                                  //    (template)
} ///:~
```

```
template<typename T>
const T& min(const T& a, const T& b, const T& c);
```

6.2.3.

```
//: C05:TemplateFunctionAddress.cpp {-mwcc}
// Taking the address of a function generated
// from a template.

template<typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template<typename T> void g(void (*pf)(T*)) {}

int main() {
  h(&f<int>); // Full type specification
  h(&f); // Type deduction
  g<int>(&f<int>); // Full type specification
  g(&f<int>); // Type deduction
  g<int>(&f); // Partial (but sufficient) specification
} ///:~
```

```
// The variable s is a std::string
transform(s.begin(), s.end(), s.begin(), tolower);
```

```
//: C05:FailedTransform.cpp {-xo}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s("LOWER");
  transform(s.begin(), s.end(), s.begin(), tolower);
  cout << s << endl;
} ///:~
```

```
template<class charT> charT toupper(charT c,
                                    const locale& loc);
template<class charT> charT tolower(charT c,
                                    const locale& loc);
```

```
transform(s.begin(),s.end(),s.begin()
    static_cast<int(*)(int)>(tolower));
```

```
//: C05:StrTolower.cpp {O} {-mwcc}
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;
```

```
string strTolower(string s) {
  transform(s.begin(), s.end(), s.begin(), tolower);
  return s;
} ///:~
```

```
//: C05:Tolower.cpp {-mwcc}
//{L} StrTolower
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
string strTolower(string);

int main() {
  string s("LOWER");
  cout << strTolower(s) << endl;
} ///:~
```

```
//: C05:ToLower2.cpp {-mwcc}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

template<class charT> charT strTolower(charT c) {
  return tolower(c);  // One-arg version called
}

int main() {
  string s("LOWER");
  transform(s.begin(),s.end(),s.begin(),&strTolower<char>);
  cout << s << endl;
} ///:~
```

6.2.4.

```
//: C05:ApplySequence.h
// Apply a function to an STL sequence container.

// const, 0 arguments, any type of return value:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)() const) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)();
}
```

```
// const, 1 argument, any type of return value:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A) const, A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a);
}

// const, 2 arguments, any type of return value:
template<class Seq, class T, class R,
         class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2) const,
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a1, a2);
}

// Non-const, 0 arguments, any type of return value:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)()) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)();
}

// Non-const, 1 argument, any type of return value:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A), A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a);
}

// Non-const, 2 arguments, any type of return value:
template<class Seq, class T, class R,
         class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a1, a2);
}
// Etc., to handle maximum likely arguments ///:~
```

```
//: C05:Gromit.h
// The techno-dog. Has member functions
// with various numbers of arguments.
#include <iostream>

class Gromit {
  int arf;
  int totalBarks;
public:
  Gromit(int arf = 1) : arf(arf + 1), totalBarks(0) {}
  void speak(int) {
```

```
    for(int i = 0; i < arf; i++) {
      std::cout << "arf! ";
      ++totalBarks;
    }
    std::cout << std::endl;
  }
  char eat(float) const {
    std::cout << "chomp!" << std::endl;
    return 'z';
  }
  int sleep(char, double) const {
    std::cout << "zzz..." << std::endl;
    return 0;
  }
  void sit() const {
    std::cout << "Sitting..." << std::endl;
  }
}; ///:~
```

```
//: C05:ApplyGromit.cpp
// Test ApplySequence.h.
#include <cstddef>
#include <iostream>
#include <vector>
#include "ApplySequence.h"
#include "Gromit.h"
#include "../purge.h"
using namespace std;

int main() {
  vector<Gromit*> dogs;
  for(size_t i = 0; i < 5; i++)
    dogs.push_back(new Gromit(i));
  apply(dogs, &Gromit::speak, 1);
  apply(dogs, &Gromit::eat, 2.0f);
  apply(dogs, &Gromit::sleep, 'z', 3.0);
  apply(dogs, &Gromit::sit);
  purge(dogs);
} ///:~
```

6.2.5.

```
template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);
```

```
//: C05:PartialOrder.cpp
// Reveals ordering of function templates.
#include <iostream>
using namespace std;
```

```
template<class T> void f(T) {
  cout << "T" << endl;
}

template<class T> void f(T*) {
  cout << "T*" << endl;
}

template<class T> void f(const T*) {
  cout << "const T*" << endl;
}

int main() {
  f(0);            // T
  int i = 0;
  f(&i);           // T*
  const int j = 0;
  f(&j);           // const T*
} ///:~
```

6.3.

6.3.1.

```
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
```

```
//: C05:MinTest2.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<class T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}

// An explicit specialization of the min template
template<>
const char* const& min<const char*>(const char* const& a,
                                    const char* const& b) {
  return (strcmp(a, b) < 0) ? a : b;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(s1, s2) << endl;
  cout << min<>(s1, s2) << endl;
} ///:~
```

```
template<class T, class Allocator = allocator<T> >
class vector {...};
```

```
template<> class vector<bool, allocator<bool> > {...};
```

6.3.2.

```
template<class Allocator> class vector<bool, Allocator>;
```

```
//: C05:PartialOrder2.cpp
// Reveals partial ordering of class templates.
#include <iostream>
using namespace std;

template<class T, class U> class C {
public:
  void f() { cout << "Primary Template\n"; }
};

template<class U> class C<int, U> {
public:
  void f() { cout << "T == int\n"; }
};

template<class T> class C<T, double> {
public:
  void f() { cout << "U == double\n"; }
};

template<class T, class U> class C<T*, U> {
public:
  void f() { cout << "T* used\n"; }
};

template<class T, class U> class C<T, U*> {
public:
  void f() { cout << "U* used\n"; }
};

template<class T, class U> class C<T*, U*> {
public:
  void f() { cout << "T* and U* used\n"; }
};

template<class T> class C<T, T> {
public:
  void f() { cout << "T == U\n"; }
};

int main() {
  C<float, int>().f();    // 1: Primary template
  C<int, float>().f();    // 2: T == int
```

```
  C<float, double>().f(); // 3: U == double
  C<float, float>().f();  // 4: T == U
  C<float*, float>().f(); // 5: T* used [T is float]
  C<float, float*>().f(); // 6: U* used [U is float]
  C<float*, int*>().f();  // 7: T* and U* used [float,int]
  // The following are ambiguous:
//   8: C<int, int>().f();
//   9: C<double, double>().f();
//  10: C<float*, float*>().f();
//  11: C<int, int*>().f();
//  12: C<int*, int*>().f();
} ///:~
```

6.3.3.

```
//: C05:Sortable.h
// Template specialization.
#ifndef SORTABLE_H
#define SORTABLE_H
#include <cstring>
#include <cstddef>
#include <string>
#include <vector>
using std::size_t;

template<class T>
class Sortable : public std::vector<T> {
public:
  void sort();
};

template<class T>
void Sortable<T>::sort() { // A simple sort
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(this->at(j-1) > this->at(j)) {
        T t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}

// Partial specialization for pointers:
template<class T>
class Sortable<T*> : public std::vector<T*> {
public:
  void sort();
};

template<class T>
void Sortable<T*>::sort() {
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(*this->at(j-1) > *this->at(j)) {
        T* t = this->at(j-1);
```

```
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}

// Full specialization for char*
// (Made inline here for convenience -- normally you would
// place the function body in a separate file and only
// leave the declaration here).
template<> inline void Sortable<char*>::sort() {
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(std::strcmp(this->at(j-1), this->at(j)) > 0) {
        char* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}
#endif // SORTABLE_H ///:~
```

```
//: C05:Sortable.cpp
//{-bor} (Because of bitset in Urand.h)
// Testing template specialization.
#include <cstddef>
#include <iostream>
#include "Sortable.h"
#include "Urand.h"
using namespace std;

#define asz(a) (sizeof a / sizeof a[0])

char* words[] = { "is", "running", "big", "dog", "a", };
char* words2[] = { "this", "that", "theother", };

int main() {
  Sortable<int> is;
  Urand<47> rnd;
  for(size_t i = 0; i < 15; ++i)
    is.push_back(rnd());
  for(size_t i = 0; i < is.size(); ++i)
    cout << is[i] << ' ';
  cout << endl;
  is.sort();
  for(size_t i = 0; i < is.size(); ++i)
    cout << is[i] << ' ';
  cout << endl;

  // Uses the template partial specialization:
  Sortable<string*> ss;
  for(size_t i = 0; i < asz(words); ++i)
    ss.push_back(new string(words[i]));
  for(size_t i = 0; i < ss.size(); ++i)
    cout << *ss[i] << ' ';
  cout << endl;
  ss.sort();
  for(size_t i = 0; i < ss.size(); ++i) {
```

```
    cout << *ss[i] << ' ';
    delete ss[i];
  }
  cout << endl;

  // Uses the full char* specialization:
  Sortable<char*> scp;
  for(size_t i = 0; i < asz(words2); ++i)
    scp.push_back(words2[i]);
  for(size_t i = 0; i < scp.size(); ++i)
    cout << scp[i] << ' ';
  cout << endl;
  scp.sort();
  for(size_t i = 0; i < scp.size(); ++i)
    cout << scp[i] << ' ';
  cout << endl;
} ///:~
```

6.3.4.

```
//: C05:DelayedInstantiation.cpp
// Member functions of class templates are not
// instantiated until they're needed.

class X {
public:
  void f() {}
};

class Y {
public:
  void g() {}
};

template<typename T> class Z {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
};

int main() {
  Z<X> zx;
  zx.a(); // Doesn't create Z<X>::b()
  Z<Y> zy;
  zy.b(); // Doesn't create Z<Y>::a()
} ///:~
```

```
//: C05:Nobloat.h
// Shares code for storing pointers in a Stack.
#ifndef NOBLOAT_H
#define NOBLOAT_H
```

```
#include <cassert>
#include <cstddef>
#include <cstring>

// The primary template
template<class T> class Stack {
  T* data;
  std::size_t count;
  std::size_t capacity;
  enum { INIT = 5 };
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new T[INIT];
  }
  void push(const T& t) {
    if(count == capacity) {
      // Grow array store
      std::size_t newCapacity = 2 * capacity;
      T* newData = new T[newCapacity];
      for(size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
    --count;
  }
  T top() const {
    assert(count > 0);
    return data[count-1];
  }
  std::size_t size() const { return count; }
};

// Full specialization for void*
template<> class Stack<void *> {
  void** data;
  std::size_t count;
  std::size_t capacity;
  enum { INIT = 5 };
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new void*[INIT];
  }
  void push(void* const & t) {
    if(count == capacity) {
      std::size_t newCapacity = 2*capacity;
      void** newData = new void*[newCapacity];
      std::memcpy(newData, data, count*sizeof(void*));
```

```
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
    --count;
  }
  void* top() const {
    assert(count > 0);
    return data[count-1];
  }
  std::size_t size() const { return count; }
};

// Partial specialization for other pointer types
template<class T> class Stack<T*> : private Stack<void *> {
  typedef Stack<void *> Base;
public:
  void push(T* const & t) { Base::push(t); }
  void pop() {Base::pop();}
  T* top() const { return static_cast<T*>(Base::top()); }
  std::size_t size() { return Base::size(); }
};
#endif // NOBLOAT_H ///:~
```

```
//: C05:NobloatTest.cpp
#include <iostream>
#include <string>
#include "Nobloat.h"
using namespace std;

template<class StackType>
void emptyTheStack(StackType& stk) {
  while(stk.size() > 0) {
    cout << stk.top() << endl;
    stk.pop();
  }
}

// An overload for emptyTheStack (not a specialization!)
template<class T>
void emptyTheStack(Stack<T*>& stk) {
  while(stk.size() > 0) {
    cout << *stk.top() << endl;
    stk.pop();
  }
}

int main() {
  Stack<int> s1;
  s1.push(1);
  s1.push(2);
```

```
  emptyTheStack(s1);
  Stack<int *> s2;
  int i = 3;
  int j = 4;
  s2.push(&i);
  s2.push(&j);
  emptyTheStack(s2);
} ///:~
```

6.4.

6.4.1.

```
MyClass::f();
x.f();
p->f();
```

```
#include <iostream>
#include <string>
// ...
  std::string s("hello");
  std::cout << s << std::endl;
```

```
std::operator<<(std::operator<<(std::cout,s),std::endl);
```

```
operator<<(std::cout, s);
```

```
(f)(x, y); // ADL suppressed
```

```
//: C05:Lookup.cpp
// Only produces correct behavior with EDG,
// and Metrowerks using a special option.
#include <iostream>
using std::cout;
using std::endl;

void f(double) { cout << "f(double)" << endl; }

template<class T> class X {
public:
  void g() { f(1); }
};

void f(int) { cout << "f(int)" << endl; }

int main() {
  X<int>().g();
```

```
} ///:~
```

```
f(double)
```

```
//: C05:Lookup2.cpp {-bor}{-g++}{-dmc}
// Microsoft: use option -Za (ANSI mode)
#include <algorithm>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

void g() { cout << "global g()" << endl; }

template<class T> class Y {
public:
  void g() {
    cout << "Y<" << typeid(T).name() << ">::g()" << endl;
  }
  void h() {
    cout << "Y<" << typeid(T).name() << ">::h()" << endl;
  }
  typedef int E;
};

typedef double E;

template<class T> void swap(T& t1, T& t2) {
  cout << "global swap" << endl;
  T temp = t1;
  t1 = t2;
  t2 = temp;
}

template<class T> class X : public Y<T> {
public:
  E f() {
    g();
    this->h();
    T t1 = T(), t2 = T(1);
    cout << t1 << endl;
    swap(t1, t2);
    std::swap(t1, t2);
    cout << typeid(E).name() << endl;
    return E(t2);
  }
};

int main() {
  X<int> x;
  cout << x.f() << endl;
} ///:~
```

```
global g()
Y<int>::h()
0
global swap
double
1
```

6.4.2.

```
//: C05:FriendScope.cpp
#include <iostream>
using namespace std;

class Friendly {
  int i;
public:
  Friendly(int theInt) { i = theInt; }
  friend void f(const Friendly&); // Needs global def.
  void g() { f(*this); }
};

void h() {
  f(Friendly(1));  // Uses ADL
}

void f(const Friendly& fo) {  // Definition of friend
  cout << fo.i << endl;
}

int main() {
  h(); // Prints 1
  Friendly(2).g(); // Prints 2
} ///:~
```

```
//: C05:FriendScope2.cpp
#include <iostream>
using namespace std;

// Necessary forward declarations:
template<class T> class Friendly;
template<class T> void f(const Friendly<T>&);

template<class T> class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f<>(const Friendly<T>&);
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}
```

```
template<class T> void f(const Friendly<T>& fo) {
  cout << fo.t << endl;
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

```
//: C05:FriendScope3.cpp {-bor}
// Microsoft: use the -Za (ANSI-compliant) option
#include <iostream>
using namespace std;

template<class T> class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f(const Friendly<T>& fo) {
    cout << fo.t << endl;
  }
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

```
template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
};
```

```
//: C05:Box1.cpp
// Defines template operators.
#include <iostream>
using namespace std;

// Forward declarations
template<class T> class Box;

template<class T>
Box<T> operator+(const Box<T>&, const Box<T>&);

template<class T>
ostream& operator<<(ostream&, const Box<T>&);
```

```
template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
  friend Box operator+<>(const Box<T>&, const Box<T>&);
  friend ostream& operator<< <>(ostream&, const Box<T>&);
};

template<class T>
Box<T> operator+(const Box<T>& b1, const Box<T>& b2) {
  return Box<T>(b1.t + b2.t);
}

template<class T>
ostream& operator<<(ostream& os, const Box<T>& b) {
  return os << '[' << b.t << ']';
}

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl;  // [3]
//  cout << b1 + 2 << endl; // No implicit conversions!
} ///:~
```

```
//: C05:Box2.cpp
// Defines non-template operators.
#include <iostream>
using namespace std;

template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
  friend Box<T> operator+(const Box<T>& b1,
                          const Box<T>& b2) {
    return Box<T>(b1.t + b2.t);
  }
  friend ostream&
  operator<<(ostream& os, const Box<T>& b) {
    return os << '[' << b.t << ']';
  }
};

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl; // [3]
  cout << b1 + 2 << endl; // [3]
} ///:~
```

```
// Inside Friendly:
friend void f<>(const Friendly<double>&);
```

```
// Inside Friendly:
friend void g(int); // g(int) befriends all Friendlys
```

```
template<class T> class Friendly {
  template<class U> friend void f<>(const Friendly<U>&);
```

6.5.

6.5.1.

```
template<class T> class numeric_limits {
public:
  static const bool is_specialized = false;
  static T min() throw();
  static T max() throw();
  static const int digits = 0;
  static const int digits10 = 0;
  static const bool is_signed = false;
  static const bool is_integer = false;
  static const bool is_exact = false;
  static const int radix = 0;
  static T epsilon() throw();
  static T round_error() throw();
  static const int min_exponent = 0;
  static const int min_exponent10 = 0;
  static const int max_exponent = 0;
  static const int max_exponent10 = 0;
  static const bool has_infinity = false;
  static const bool has_quiet_NaN = false;
  static const bool has_signaling_NaN = false;
  static const float_denorm_style has_denorm =
                                  denorm_absent;
  static const bool has_denorm_loss = false;
  static T infinity() throw();
  static T quiet_NaN() throw();
  static T signaling_NaN() throw();
  static T denorm_min() throw();
  static const bool is_iec559 = false;
  static const bool is_bounded = false;
  static const bool is_modulo = false;
  static const bool traps = false;
  static const bool tinyness_before = false;
  static const float_round_style round_style =
                                 round_toward_zero;
};
```

```
template<class charT,
  class traits = char_traits<charT>,
  class allocator = allocator<charT> >
  class basic_string;
```

```
template<> struct char_traits<char> {
  typedef char char_type;
  typedef int int_type;
  typedef streamoff off_type;
  typedef streampos pos_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char_type& c1, const char_type& c2);
  static int compare(const char_type* s1,
                     const char_type* s2, size_t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s,
                               size_t n,
                               const char_type& a);
  static char_type* move(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* copy(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n,
                           char_type a);
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,
                          const int_type& c2);
  static int_type eof();
};
```

```
std::string s;
```

```
std::basic_string<char, std::char_traits<char>,
   std::allocator<char> > s;
```

```
//: C05:BearCorner.h
#ifndef BEARCORNER_H
#define BEARCORNER_H
#include <iostream>
using std::ostream;

// Item classes (traits of guests):
class Milk {
public:
  friend ostream& operator<<(ostream& os, const Milk&) {
    return os << "Milk";
  }
};

class CondensedMilk {
public:
  friend ostream&
  operator<<(ostream& os, const CondensedMilk &) {
    return os << "Condensed Milk";
```

```
  }
};

class Honey {
public:
  friend ostream& operator<<(ostream& os, const Honey&) {
    return os << "Honey";
  }
};

class Cookies {
public:
  friend ostream& operator<<(ostream& os, const Cookies&) {
    return os << "Cookies";
  }
};

// Guest classes:
class Bear {
public:
  friend ostream& operator<<(ostream& os, const Bear&) {
    return os << "Theodore";
  }
};

class Boy {
public:
  friend ostream& operator<<(ostream& os, const Boy&) {
    return os << "Patrick";
  }
};

// Primary traits template (empty-could hold common types)
template<class Guest> class GuestTraits;

// Traits specializations for Guest types
template<> class GuestTraits<Bear> {
public:
  typedef CondensedMilk beverage_type;
  typedef Honey snack_type;
};

template<> class GuestTraits<Boy> {
public:
  typedef Milk beverage_type;
  typedef Cookies snack_type;
};
#endif // BEARCORNER_H ///:~
```

6.5.2.

```
template<> struct char_traits<wchar_t> {
  typedef wchar_t char_type;
  typedef wint_t int_type;
  typedef streamoff off_type;
```

```
  typedef wstreampos pos_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char_type& c1, const char_type& c2);
  static int compare(const char_type* s1,
                     const char_type*  s2, size_t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s,
                               size_t n,
                               const char_type& a);
  static char_type* move(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* copy(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n,
                           char_type a);
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,
                          const int_type& c2);
  static int_type eof();
};
```

```
//: C05:BearCorner2.cpp
// Illustrates policy classes.
#include <iostream>
#include "BearCorner.h"
using namespace std;

// Policy classes (require a static doAction() function):
class Feed {
public:
  static const char* doAction() { return "Feeding"; }
};

class Stuff {
public:
  static const char* doAction() { return "Stuffing"; }
};

// The Guest template (uses a policy and a traits class)
template<class Guest, class Action,
         class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << Action::doAction() << " " << theGuest
         << " with " << bev
         << " and " << snack << endl;
```

```
  }
};

int main() {
  Boy cr;
  BearCorner<Boy, Feed> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear, Stuff> pc2(pb);
  pc2.entertain();
} ///:~
```

6.5.3.

```
//: C05:CountedClass.cpp
// Object counting via static members.
#include <iostream>
using namespace std;

class CountedClass {
  static int count;
public:
  CountedClass() { ++count; }
  CountedClass(const CountedClass&) { ++count; }
  ~CountedClass() { --count; }
  static int getCount() { return count; }
};

int CountedClass::count = 0;

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;   // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;   // 2
  { // An arbitrary scope:
    CountedClass c(b);
    cout << CountedClass::getCount() << endl; // 3
    a = c;
    cout << CountedClass::getCount() << endl; // 3
  }
  cout << CountedClass::getCount() << endl;   // 2
} ///:~
```

```
//: C05:CountedClass2.cpp
// Erroneous attempt to count objects.
#include <iostream>
using namespace std;

class Counted {
  static int count;
public:
```

```
  Counted() { ++count; }
  Counted(const Counted&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};

int Counted::count = 0;

class CountedClass : public Counted {};
class CountedClass2 : public Counted {};

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 3 (Error)
} ///:~
```

```
//: C05:CountedClass3.cpp
#include <iostream>
using namespace std;

template<class T> class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted<T>&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};

template<class T> int Counted<T>::count = 0;

// Curious class definitions
class CountedClass : public Counted<CountedClass> {};
class CountedClass2 : public Counted<CountedClass2> {};

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 1 (!)
} ///:~
```

6.6.

```
//: C05:Factorial.cpp
// Compile-time computation using templates.
```

```
#include <iostream>
using namespace std;

template<int n> struct Factorial {
  enum { val = Factorial<n-1>::val * n };
};

template<> struct Factorial<0> {
  enum { val = 1 };
};

int main() {
  cout << Factorial<12>::val << endl; // 479001600
} ///:~
```

```
double nums[Factorial<5>::val];
assert(sizeof nums == sizeof(double)*120);
```

6.6.1.

```
//: C05:Fibonacci.cpp
#include <iostream>
using namespace std;

template<int n> struct Fib {
  enum { val = Fib<n-1>::val + Fib<n-2>::val };
};

template<> struct Fib<1> { enum { val = 1 }; };

template<> struct Fib<0> { enum { val = 0 }; };

int main() {
  cout << Fib<5>::val << endl;   // 6
  cout << Fib<20>::val << endl;  // 6765
} ///:~
```

```
int val = 1;
while(p--)
   val *= n; -->
```

```
int power(int n, int p) {
  return (p == 0) ? 1 : n*power(n, p - 1);
}
```

```
//: C05:Power.cpp
#include <iostream>
using namespace std;
```

```
template<int N, int P> struct Power {
  enum { val = N * Power<N, P-1>::val };
};

template<int N> struct Power<N, 0> {
  enum { val = 1 };
};

int main() {
  cout << Power<2, 5>::val << endl;  // 32
} ///:~
```

```
//: C05:Accumulate.cpp
// Passes a "function" as a parameter at compile time.
#include <iostream>
using namespace std;

// Accumulates the results of F(0)..F(n)
template<int n, template<int> class F> struct Accumulate {
  enum { val = Accumulate<n-1, F>::val + F<n>::val };
};

// The stopping criterion (returns the value F(0))
template<template<int> class F> struct Accumulate<0, F> {
  enum { val = F<0>::val };
};

// Various "functions":
template<int n> struct Identity {
  enum { val = n };
};

template<int n> struct Square {
  enum { val = n*n };
};

template<int n> struct Cube {
  enum { val = n*n*n };
};

int main() {
  cout << Accumulate<4, Identity>::val << endl; // 10
  cout << Accumulate<4, Square>::val << endl;   // 30
  cout << Accumulate<4, Cube>::val << endl;     // 100
} ///:~
```

```
void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
  for(int i = 0; i < ROWS; ++i) {
      y[i] = 0;
      for(int j = 0; j < COLS; ++j)
```

```
        y[i] += a[i][j]*x[j];
  }
}
```

```
void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
  for(int i = 0; i < ROWS; ++i) {
      y[i] = 0;
      for(int j = 0; j < COLS; j += 2)
        y[i] += a[i][j]*x[j] + a[i][j+1]*x[j+1];
  }
}
```

```
//: C05:Unroll.cpp
// Unrolls an implicit loop via inlining.
#include <iostream>
using namespace std;

template<int n> inline int power(int m) {
  return power<n-1>(m) * m;
}

template<> inline int power<1>(int m) {
  return m;
}

template<> inline int power<0>(int m) {
  return 1;
}

int main() {
  int m = 4;
  cout << power<3>(m) << endl;
} ///:~
```

```
//: C05:Max.cpp
#include <iostream>
using namespace std;

template<int n1, int n2> struct Max {
  enum { val = n1 > n2 ? n1 : n2 };
};

int main() {
  cout << Max<10, 20>::val << endl;  // 20
} ///:~
```

```
//: C05:Conditionals.cpp
// Uses compile-time conditions to choose code.
```

```
#include <iostream>
using namespace std;

template<bool cond> struct Select {};

template<> class Select<true> {
  static void statement1() {
    cout << "This is statement1 executing\n";
  }
public:
  static void f() { statement1(); }
};

template<> class Select<false> {
  static void statement2() {
    cout << "This is statement2 executing\n";
  }
public:
  static void f() { statement2(); }
};

template<bool cond> void execute() {
  Select<cond>::f();
}

int main() {
  execute<sizeof(int) == 4>();
} ///:~
```

```
if(cond)
  statement1();
else
  statement2();
```

```
//: C05:StaticAssert1.cpp {-xo}
// A simple, compile-time assertion facility

#define STATIC_ASSERT(x) \
  do { typedef int a[(x) ? 1 : -1]; } while(0)

int main() {
  STATIC_ASSERT(sizeof(int) <= sizeof(long)); // Passes
  STATIC_ASSERT(sizeof(double) <= sizeof(int)); // Fails
} ///:~
```

```
//: C05:StaticAssert2.cpp {-g++}
#include <iostream>
using namespace std;

// A template and a specialization
```

```
template<bool> struct StaticCheck {
  StaticCheck(...);
};

template<> struct StaticCheck<false> {};

// The macro (generates a local class)
#define STATIC_CHECK(expr, msg) {              \
  class Error_##msg {};                        \
  sizeof((StaticCheck<expr>(Error_##msg()))); \
}

// Detects narrowing conversions
template<class To, class From> To safe_cast(From from) {
  STATIC_CHECK(sizeof(From) <= sizeof(To),
               NarrowingConversion);
  return reinterpret_cast<To>(from);
}

int main() {
  void* p = 0;
  int i = safe_cast<int>(p);
  cout << "int cast okay" << endl;
  //! char c = safe_cast<char>(p);
} ///:~
```

```
int i = safe_cast<int>(p);
```

```
{                                                    \
   class Error_NarrowingConversion {};               \
   sizeof(StaticCheck<sizeof(void*) <= sizeof(int)> \
           (Error_NarrowingConversion()));           \
}
```

```
char c = safe_cast<char>(p);
```

```
{                                                     \
   class Error_NarrowingConversion {};                \
   sizeof(StaticCheck<sizeof(void*) <= sizeof(char)> \
           (Error_NarrowingConversion()));            \
}
```

```
sizeof(StaticCheck<false>(Error_NarrowingConversion()));
```

```
Cannot cast from 'Error_NarrowingConversion' to 'StaticCheck<0>' in function
char safe_cast<char,void *>(void *)
```

6.6.2.

```
D = A + B + C;
```

```
//: C05:MyVector.cpp
// Optimizes away temporaries via templates.
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// A proxy class for sums of vectors
template<class, size_t> class MyVectorSum;

template<class T, size_t N> class MyVector {
  T data[N];
public:
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for(size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  MyVector<T,N>& operator=(const MyVectorSum<T,N>& right);
  const T& operator[](size_t i) const { return data[i]; }
  T& operator[](size_t i) { return data[i]; }
};

// Proxy class hold references; uses lazy addition
template<class T, size_t N> class MyVectorSum {
  const MyVector<T,N>& left;
  const MyVector<T,N>& right;
public:
  MyVectorSum(const MyVector<T,N>& lhs,
              const MyVector<T,N>& rhs)
  : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
  }
};

// Operator to support v3 = v1 + v2
template<class T, size_t N> MyVector<T,N>&
MyVector<T,N>::operator=(const MyVectorSum<T,N>& right) {
  for(size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}

// operator+ just stores references
template<class T, size_t N> inline MyVectorSum<T,N>
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N>(left, right);
}
```

```
// Convenience functions for the test program below
template<class T, size_t N> void init(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
    v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}

int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  MyVector<int, 5> v4;
  // Not yet supported:
//!  v4 = v1 + v2 + v3;
} ///:~
```

```
v1 = v2 + v3; // Add two vectors
```

```
v3.operator=<int,5>(MyVectorSum<int,5>(v2, v3));
```

```
v4 = v1 + v2 + v3;
```

```
(v1 + v2) + v3;
```

```
//: C05:MyVector2.cpp
// Handles sums of any length with expression templates.
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// A proxy class for sums of vectors
template<class, size_t, class, class> class MyVectorSum;

template<class T, size_t N> class MyVector {
  T data[N];
public:
```

```
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for(size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  template<class Left, class Right> MyVector<T,N>&
  operator=(const MyVectorSum<T,N,Left,Right>& right);
  const T& operator[](size_t i) const {
    return data[i];
  }
  T& operator[](size_t i) {
    return data[i];
  }
};

// Allows mixing MyVector and MyVectorSum
template<class T, size_t N, class Left, class Right>
class MyVectorSum {
  const Left& left;
  const Right& right;
public:
  MyVectorSum(const Left& lhs, const Right& rhs)
  : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
  }
};

template<class T, size_t N>
template<class Left, class Right>
MyVector<T,N>&
MyVector<T,N>::
operator=(const MyVectorSum<T,N,Left,Right>& right) {
  for(size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}
// operator+ just stores references
template<class T, size_t N>
inline MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
      (left,right);
}

template<class T, size_t N, class Left, class Right>
inline MyVectorSum<T, N, MyVectorSum<T,N,Left,Right>,
            MyVector<T,N> >
operator+(const MyVectorSum<T,N,Left,Right>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N,MyVectorSum<T,N,Left,Right>,
                        MyVector<T,N> >
    (left, right);
}
// Convenience functions for the test program below
template<class T, size_t N> void init(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
```

```
    v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}

int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  // Now supported:
  MyVector<int, 5> v4;
  v4 = v1 + v2 + v3;
  print(v4);
  MyVector<int, 5> v5;
  v5 = v1 + v2 + v3 + v4;
  print(v5);
} ///:~
```

```
v4 = v1 + v2 + v3;
```

```
v4.operator+(MVS(MVS(v1, v2), v3));
```

6.7.

6.7.1.

6.7.2.

```
//: C05:OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// The declaration of min()
template<typename T> const T& min(const T&, const T&);
#endif // OURMIN_H ///:~
```

```
//: C05:MinInstances.cpp {O}
#include "OurMin.cpp"
```

```
// Explicit Instantiations for int and double
template const int& min<int>(const int&, const int&);
template const double& min<double>(const double&,
                                   const double&);
///:~
```

```
//: C05:OurMin.cpp {O}
#ifndef OURMIN_CPP
#define OURMIN_CPP
#include "OurMin.h"

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
#endif // OURMIN_CPP ///:~
```

```
1
3.1
```

6.7.3.

```
//: C05:OurMin2.h
// Declares min as an exported template
// (Only works with EDG-based compilers)
#ifndef OURMIN2_H
#define OURMIN2_H
export template<typename T> const T& min(const T&,
                                         const T&);
#endif // OURMIN2_H ///:~
```

```
// C05:OurMin2.cpp
// The definition of the exported min template
// (Only works with EDG-based compilers)
#include "OurMin2.h"
export
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
} ///:~
```

6.7.4.

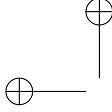
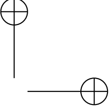
6.8.

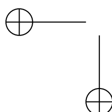
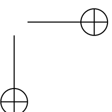
```
//: C05:Exercise4.cpp {-xo}
class Noncomparable {};
```

```
struct HardLogic {
  Noncomparable nc1, nc2;
  void compare() {
    return nc1 == nc2; // Compiler error
  }
};

template<class T> struct SoftLogic {
  Noncomparable nc1, nc2;
  void noOp() {}
  void compare() {
    nc1 == nc2;
  }
};

int main() {
  SoftLogic<Noncomparable> l;
  l.noOp();
} ///:~
```

```
//: C05:Exercise7.cpp {-xo}
class Buddy {};

template<class T> class My {
  int i;
public:
  void play(My<Buddy>& s) {
    s.i = 3;
  }
};

int main() {
  My<int> h;
  My<Buddy> me, bud;
  h.play(bud);
  me.play(bud);
} ///:~
```

```
//: C05:Exercise8.cpp {-xo}
template<class T> double pythag(T a, T b, T c) {
  return (-b + sqrt(double(b*b - 4*a*c))) / 2*a;
}

int main() {
  pythag(1, 2, 3);
  pythag(1.0, 2.0, 3.0);
  pythag(1, 2.0, 3.0);
  pythag<double>(1, 2.0, 3.0);
} ///:~
```

7: Algoritmos genéricos

Los algoritmos son la base de la computación. Ser capaz de escribir un algoritmo que funcione con cualquier tipo de se secuencia hace que sus programas sean simples y seguros. La habilidad para adaptar algoritmos en tiempo de ejecución a revolucionado el desarrollo de software.

El subconjunto de la Librería Estándar de C++ conocido como Standard Template Library (STL)[1] fue diseñado entorno a algoritmos genéricos —código que procesa secuencias de cualquier tipo de valores de un modo seguro. El objetivo era usar algoritmos predefinidos para casi cualquier tarea, en lugar de codificar a mano cada vez que se necesitara procesar una colección de datos. Sin embargo, ese potencial requiere cierto aprendizaje. Para cuando llegue al final de este capítulo, debería ser capaz de decidir por sí mismo si los algoritmos le resultan útiles o demasiado confusos de recordar. Si es como la mayoría de la gente, se resistirá al principio pero entonces tenderá a usarlos más y más con el tiempo.

7.1. Un primer vistazo

Entre otras cosas, los algoritmos genéricos de la librería estándar proporcionan un vocabulario con el que desribir soluciones. Una vez que los algoritmos le sean familiares, tendrá un nuevo conjunto de palabras con el que discutir que está haciendo, y esas palabras son de un nivel mayor que las que tenía antes. No necesitará decir «Este bucle recorre y asigna de aquí a ahí... oh, ya veo, ¡está copiando!» En su lugar dirá simplemente `copy()`. Esto es lo que hemos estado haciendo desde el principio de la programación de computadores —creando abstracciones de alto nivel para expresar lo que está haciendo y perder menos tiempo diciendo cómo hacerlo. El «cómo» se ha resuelto una vez y para todo y está oculto en el código del algoritmo, listo para ser reutilizado cuando se necesite.

Vea aquí un ejemplo de cómo utilizar el algoritmo `copy`:

```
//: C06:CopyInts.cpp
// Copies ints without an explicit loop.
#include <algorithm>
#include <cassert>
#include <cstddef>  // For size_t
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
```

[1] N. de T.: Librería Estándar de Plantillas.

```
  copy(a, a + SIZE, b);
  for(size_t i = 0; i < SIZE; ++i)
    assert(a[i] == b[i]);
} ///:~
```

Los dos primeros parámetros de `copy` representan el rango de la secuencia de entrada —en este caso del array `a`. Los rangos se especifican con un par de punteros. El primero apunta al primer elemento de la secuencia, y el segungo apunta una posición después del final del array (justo después del último elemento). Esto puede parecer extraño al principio, pero es una antigua expresión idiomática de C que resulta bastante práctica. Por ejemplo, la diferencia entre esos dos punteros devuelve el número de elementos de la secuencia. Más importante, en la implementación de `copy()`, el segundo puntero puede actual como un centinela para para la iteración a través de la secuencia. El tercer argumento hace referencia al comienzo de la secuencia de salida, que es el array `b` en el ejemplo. Se asume que el array `b` tiene suficiente espacio para recibir los elementos copiados.

El algotirmo `copy()` no parece muy excitante if solo puediera procesar enteros. Puede copiar cualquier tipo de secuencia. El siguiente ejemplo copia objetos `string`.

```
//: C06:CopyStrings.cpp
// Copies strings.
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <string>
using namespace std;

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  copy(a, a + SIZE, b);
  assert(equal(a, a + SIZE, b));
} ///:~
```

Este ejmeplo presenta otro algoritmo, `equal()`, que devuelve cierto solo si cada elemento de la primera secuencia es igual (usando su `operator==()`) a su elemento correspondiente en la segunda secuencia. Este ejemplo recorre cada secuencia 2 veces, una para copiar, y otra para comparar, sin ningún bucle explícito.

Los algoritmos genéricos consiguen esta flexibilidad porque son funciones parametrizadas (plantillas). Si piensa en la implementación de `copy()` verá que es algo como lo siguiente, que es «casi» correcto:

```
template<typename T>
void copy(T* begin, T* end, T* dest) {
  while (begin != end)
    *dest++ = *begin++;
}
```

Decimos «casi» porque `copy()` puede procesar secuencias delimitadas por cual-

quier cosa que actúe como un puntero, tal como un iterador. De ese modo, `copy()` se puede utilizar para duplicar un `vector`, como en el siguiente ejemplo.

```
//: C06:CopyVector.cpp
// Copies the contents of a vector.
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <vector>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2(SIZE);
  copy(v1.begin(), v1.end(), v2.begin());
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

El primer vector, `v1`, es inicializado a partir de una secuencia de enteros en el array `a`. La definición del vector `v2` usa un contructor diferente de `vector` que reserva sitio para `SIZE` elementos, inicializados a cero (el valor por defecto para enteros).

Igual que con el ejemplo anterior con el array, es importante que `v2` tenga suficiente espacio para recibir una copia de los contenidos de `v1`. Por conveniencia, una función de librería especial, `back_inserter()`, retorna un tipo especial de iterador que inserta elementos en lugar de sobre-escribirlos, de modo que la memoria del contenedor se expande conforme se necesita. El siguiente ejemplo usa `back_inserter()`, y por eso no hay que establecer el tamaño del vector de salida, `v2`, antes de tiempo.

```
//: C06:InsertVector.cpp
// Appends the contents of a vector to another.
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <iterator>
#include <vector>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2;  // v2 is empty here
  copy(v1.begin(), v1.end(), back_inserter(v2));
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

La función `back_inserter()` está definida en el fichero de cabecera `<iterator>`. Explicaremos los iteradores de inserción en profundidad en el próximo capítulo.

Dado que los iteradores son idénticos a punteros en todos los sentidos importan-

tes, puede escribir los algoritmos de la librería estándar de modo que los argumentos puedan ser tanto punteros como iteradores. Por esta razón, la implementación de c-opy() se parece más al siguiente código:

```
template<typename Iterator>
void copy(Iterator begin, Iterator end, Iterator dest) {
  while (begin != end)
    *begin++ = *dest++;
}
```

Para cualquier tipo de argumento que use en la llamada, copy() asume que implementa adecuadamente la indirección y los operadores de incremento. Si no lo hace, obtendrás un error de compilación.

7.1.1. Predicados

A veces, podría querer copiar solo un subconjunto bien definido de una secuencia a otra; solo aquellos elementos que satisfagan una condición particular. Para conseguir esta flexibilidad, muchos algoritmos tienen una forma alternativa de llamada que permite proporcionar un predicado, que es simplemente una función que retorna un valor booleano basado en algún criterio. Suponga por ejemplo, que solo quiere extraer de una secuencia de enteros, aquellos que son menores o iguales de 15. Una versión de copy() llamada remove_copy_if() puede hacer el trabajo, tal que así:

```
//: C06:CopyInts2.cpp
// Ignores ints that satisfy a predicate.
#include <algorithm>
#include <cstddef>
#include <iostream>
using namespace std;

// You supply this predicate
bool gt15(int x) { return 15 < x; }

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
  int* endb = remove_copy_if(a, a+SIZE, b, gt15);
  int* beginb = b;
  while(beginb != endb)
    cout << *beginb++ << endl; // Prints 10 only
} ///:~
```

La función remove_copy_if() acepta los rangos definidos por punteros habituales, seguidos de un predicado de su elección. El predicado debe ser un puntero a función[FIXME] que toma un argumento simple del mismo tipo que los elementos de la secuencia, y que debe retornar un booleano. Aquí, la función gt15 returna verdadero si su argumento es mayor que 15. El algoritmo remove_copy_if() aplica gt15() a cada elemento en la secuencia de entrada e ignora aquellos elementos para los cuales el predicado devuelve verdad cuando escribe la secuencia de salida.

El siguiente programa ilustra otra variación más del algoritmo de copia.

```
//: C06:CopyStrings2.cpp
// Replaces strings that satisfy a predicate.
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

// The predicate
bool contains_e(const string& s) {
  return s.find('e') != string::npos;
}

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  string* endb = replace_copy_if(a, a + SIZE, b,
    contains_e, string("kiss"));
  string* beginb = b;
  while(beginb != endb)
    cout << *beginb++ << endl;
} ///:~
```

En lugar de simplemente ignorar elementos que no satisfagan el predicado, `replace_copy_if()` substituye un valor fijo para esos elementos cuando escribe la secuencia de salida. La salida es:

```
kiss
my
lips
```

como la ocurrencia original de «read», la única cadena de entrada que contiene la letra «e», es reemplazada por la palabra «kiss», como se especificó en el último argumento en la llamada a `replace_copy_if()`.

El algoritmo `replace_if()` cambia la secuencia original in situ, en lugar de escribir en una secuencia de salida separada, tal como muestra el siguiente programa:

```
//: C06:ReplaceStrings.cpp
// Replaces strings in-place.
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

bool contains_e(const string& s) {
  return s.find('e') != string::npos;
}

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  replace_if(a, a + SIZE, contains_e, string("kiss"));
```

```
  string* p = a;
  while(p != a + SIZE)
    cout << *p++ << endl;
} ///:~
```

7.1.2. Iteradores de flujo

Como cualquier otra buena librería, la Librería Estándar de C++ intenta proporcionar modos convenientes de automatizar tareas comunes. Mencionamos al principio de este capítulo puede usar algoritmos genéricos en lugar de bucles. Hasta el momento, sin embargo, nuestros ejemplos siguen usando un bucle explícito para imprimir su salida. Dado que imprimir la salida es una de las tareas más comunes, es de esperar que haya una forma de automatizar eso también.

Ahí es donde los iteradores de flujo entran en juego. Un iterador de flujo usa un flujo como secuencia de entrada o salida. Para eliminar el bucle de salida en el programa `CopyInts2.cpp`, puede hacer algo como lo siguiente:

```
//: C06:CopyInts3.cpp
// Uses an output stream iterator.
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <iterator>
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout, "\n"), gt15);
} ///:~
```

En este ejemplo, reemplazaremos la secuencia de salida `b` en el tercer argumento de `remove_copy_if()` con un iterador de flujo de salida, que es una instancia de la clase `ostream_iterator` declarada en el fichero `<iterator>`. Los iteradores de flujo de salida sobrecargan sus operadores de copia-asignación para escribir a sus flujos. Esta instancia en particular de `ostream_iterator` está vinculada al flujo de salida `cout`. Cada vez que `remove_copy_if()` asigna un entero de la secuencia `a` a `cout` a través de este iterador, el iterador escribe el entero a `cout` y automáticamente escribe también una instancia de la cada de separador indicada en su segundo argumento, que en este caso contiene el carácter de nueva linea.

Es igual de fácil escribir en un fichero proporcionando un flujo de salida asociado a un fichero en lugar de `cout`.

```
//: C06:CopyIntsToFile.cpp
// Uses an output file stream iterator.
#include <algorithm>
#include <cstddef>
#include <fstream>
```

```
#include <iterator>
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  ofstream outf("ints.out");
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(outf, "\n"), gt15);
} ///:~
```

Un iterador de flujo de entrada permite a un algoritmo leer su secuencia de entrada desde un flujo de entrada. Esto se consigue haciendo que tanto el constructor como `operator++()` lean el siguiente elemento del flujo subyacente y sobrecargando `operator*()` para conseguir el valor leído previamente. Dado que los algoritmos requieren dos punteros para delimitar la secuencia de entrada, puede construir un `istream_iterator` de dos formas, como puede ver en el siguiente programa.

```
//: C06:CopyIntsFromFile.cpp
// Uses an input stream iterator.
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
  ofstream ints("someInts.dat");
  ints << "1 3 47 5 84 9";
  ints.close();
  ifstream inf("someInts.dat");
  assure(inf, "someInts.dat");
  remove_copy_if(istream_iterator<int>(inf),
                 istream_iterator<int>(),
                 ostream_iterator<int>(cout, "\n"), gt15);
} ///:~
```

El primer argumento de `replace_copy_if()` en este programa asocia un objeto `istream_iterator` al fichero de entrada que contiene enteros. El segundo argumento usa el constructor por defecto de la clase `istream_iterator`. Esta llamada construye un valor especial de `istream_iterator` que indica el fin de fichero, de modo que cuando el primer iterador encuentra el final del fichero físico, se compara con el valor de `istream_iterator<int>()`, permitiendo al algoritmo terminar correctamente. Fíjese que este ejemplo evita usar un array explícito.

7.1.3. Complejidad algorítmica

Usar una librería es una cuestión de confianza. Debe confiar en que los desarrolladores no solo proporcionan la funcionalidad correcta, sino también esperar que las funciones se ejecutan tan eficientemente como sea posible. Es mejor escribir sus propios bucles que usar algoritmos que degradan el rendimiento.

Para garantizar la calidad de las implementaciones de la librería, la estándar de C++ no solo especifica lo que debería hacer un algoritmo, también cómo de rápido debería hacerlo y a veces cuánto espacio debería usar. Cualquier algoritmo que no cumpla con los requisitos de rendimiento no es conforma al estándar. La medida de la eficiencia operacional de un algoritmo se llama complejidad.

Cuando es posible, el estándar especifica el número exacto de operaciones que un algoritmo debería usar. El algoritmo `count_if()`, por ejemplo, retorna el número de elementos de una secuencia que cumplan el predicado especificado. La siguiente llamada a `count_if()`, si se aplica a una secuencia de enteros similar a los ejemplos anteriores de este capítulo, devuelve el número de elementos mayores que 15:

```
size_t n = count_if(a, a + SIZE, gt15);
```

Dado que `count_if()` debe comprobar cada elemento exactamente una vez, se especificó hacer un número de comprobaciones que sea exactamente igual que el número de elementos en la secuencia. El algoritmo `copy()` tiene la misma especificación.

Otros algoritmos pueden estar especificados para realizar cierto número máximo de operaciones. El algoritmo `find()` busca a través de una secuencia hasta encontrar un elemento igual a su tercer argumento.

```
int* p = find(a, a + SIZE, 20);
```

Para tan pronto como encuentre el elemento y devuelve un puntero a la primera ocurrencia. Si no encuentra ninguno, retorna un puntero a una posición pasado el final de la secuencia (`a+SIZE` en este ejemplo). De modo que `find()` realiza como máximo tantas comparaciones como elementos tenga la secuencia.

A veces el número de operaciones que realiza un algoritmo no se puede medir con tanta precisión. En esos casos, el estándar especifica la complejidad asintótica del algoritmo, que es una medida de cómo se comportará el algoritmo con secuencias largas comparadas con formulas bien conocidas. Un buen ejemplo es el algoritmo `sort()`, del que el estándar dice que requiere «aproximadamente n log n comparaciones de media» (n es el número de elementos de la secuencia). [FIXME]. Esta medida de complejidad da una idea del coste de un algoritmo y al menos le da una base fiable para comparar algoritmos. Como verá en el siguiente capítulo, el método `find()` para el contendor `set` tiene complejidad logarítmica, que implica que el coste de una búsqueda de un elemento en un `set` será, para conjuntos grandes, proporcional al logaritmo del número de elementos. Eso es mucho menor que el número de elementos para un n grande, de modo que siempre es mejor buscar en un `set` utilizando el método en lugar del algoritmo genérico.

7.2. Objetos-función

```
//: C06:GreaterThanN.cpp
```

```
#include <iostream>
using namespace std;

class gt_n {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) { return n > value; }
};

int main() {
  gt_n f(4);
  cout << f(3) << endl;  // Prints 0 (for false)
  cout << f(5) << endl;  // Prints 1 (for true)
} ///:~
```

7.2.1. Clasificación de objetos-función

7.2.2. Creación automática de objetos-función

```
//: C06:CopyInts4.cpp
// Uses a standard function object and adaptor.
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
#include <iterator>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout, "\n"),
                 bind2nd(greater<int>(), 15));
} ///:~
```

```
//: C06:CountNotEqual.cpp
// Count elements not equal to 20.
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  cout << count_if(a, a + SIZE,
                   not1(bind1st(equal_to<int>(), 20)));// 2
} ///:~
```

7.2.3. Objetos-función adaptables

7.2.4. Más ejemplos de objetos-función

```
//: C06:Generators.h
// Different ways to fill sequences.
#ifndef GENERATORS_H
#define GENERATORS_H
#include <cstring>
#include <set>
#include <cstdlib>

// A generator that can skip over numbers:
class SkipGen {
  int i;
  int skp;
public:
  SkipGen(int start = 0, int skip = 1)
  : i(start), skp(skip) {}
  int operator()() {
    int r = i;
    i += skp;
    return r;
  }
};

// Generate unique random numbers from 0 to mod:
class URandGen {
  std::set<int> used;
  int limit;
public:
  URandGen(int lim) : limit(lim) {}
  int operator()() {
    while(true) {
      int i = int(std::rand()) % limit;
      if(used.find(i) == used.end()) {
        used.insert(i);
        return i;
      }
    }
  }
};

// Produces random characters:
class CharGen {
  static const char* source;
  static const int len;
public:
  char operator()() {
    return source[std::rand() % len];
  }
};
#endif // GENERATORS_H ///:~
```

```
//: C06:Generators.cpp {O}
```

```
#include "Generators.h"
const char* CharGen::source = "ABCDEFGHIJK"
  "LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
const int CharGen::len = std::strlen(source);
///:~
```

```
//: C06:FunctionObjects.cpp {-bor}
// Illustrates selected predefined function object
// templates from the Standard C++ library.
//{L} Generators
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

template<typename Contain, typename UnaryFunc>
void testUnary(Contain& source, Contain& dest,
  UnaryFunc f) {
  transform(source.begin(), source.end(), dest.begin(), f);
}

template<typename Contain1, typename Contain2,
  typename BinaryFunc>
void testBinary(Contain1& src1, Contain1& src2,
  Contain2& dest, BinaryFunc f) {
  transform(src1.begin(), src1.end(),
    src2.begin(), dest.begin(), f);
}

// Executes the expression, then stringizes the
// expression into the print statement:
#define T(EXPR) EXPR; print(r.begin(), r.end(), \
  "After " #EXPR);
// For Boolean tests:
#define B(EXPR) EXPR; print(br.begin(), br.end(), \
  "After " #EXPR);

// Boolean random generator:
struct BRand {
  bool operator()() { return rand() % 2 == 0; }
};

int main() {
  const int SZ = 10;
  const int MAX = 50;
  vector<int> x(SZ), y(SZ), r(SZ);
  // An integer random number generator:
  URandGen urg(MAX);
  srand(time(0));  // Randomize
  generate_n(x.begin(), SZ, urg);
```

```
  generate_n(y.begin(), SZ, urg);
  // Add one to each to guarantee nonzero divide:
  transform(y.begin(), y.end(), y.begin(),
    bind2nd(plus<int>(), 1));
  // Guarantee one pair of elements is ==:
  x[0] = y[0];
  print(x.begin(), x.end(), "x");
  print(y.begin(), y.end(), "y");
  // Operate on each element pair of x & y,
  // putting the result into r:
  T(testBinary(x, y, r, plus<int>()));
  T(testBinary(x, y, r, minus<int>()));
  T(testBinary(x, y, r, multiplies<int>()));
  T(testBinary(x, y, r, divides<int>()));
  T(testBinary(x, y, r, modulus<int>()));
  T(testUnary(x, r, negate<int>()));
  vector<bool> br(SZ); // For Boolean results
  B(testBinary(x, y, br, equal_to<int>()));
  B(testBinary(x, y, br, not_equal_to<int>()));
  B(testBinary(x, y, br, greater<int>()));
  B(testBinary(x, y, br, less<int>()));
  B(testBinary(x, y, br, greater_equal<int>()));
  B(testBinary(x, y, br, less_equal<int>()));
  B(testBinary(x, y, br, not2(greater_equal<int>())));
  B(testBinary(x,y,br,not2(less_equal<int>())));
  vector<bool> b1(SZ), b2(SZ);
  generate_n(b1.begin(), SZ, BRand());
  generate_n(b2.begin(), SZ, BRand());
  print(b1.begin(), b1.end(), "b1");
  print(b2.begin(), b2.end(), "b2");
  B(testBinary(b1, b2, br, logical_and<int>()));
  B(testBinary(b1, b2, br, logical_or<int>()));
  B(testUnary(b1, br, logical_not<int>()));
  B(testUnary(b1, br, not1(logical_not<int>())));
} ///:~
```

```
//: C06:FBinder.cpp
// Binders aren't limited to producing predicates.
//{L} Generators
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
using namespace std;

int main() {
  ostream_iterator<int> out(cout," ");
  vector<int> v(15);
  srand(time(0));  // Randomize
  generate(v.begin(), v.end(), URandGen(20));
  copy(v.begin(), v.end(), out);
  transform(v.begin(), v.end(), v.begin(),
```

```
            bind2nd(multiplies<int>(), 10));
  copy(v.begin(), v.end(), out);
} ///:~
```

```
//: C06:BinderValue.cpp
// The bound argument can vary.
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <cstdlib>
using namespace std;

int boundedRand() { return rand() % 100; }

int main() {
  const int SZ = 20;
  int a[SZ], b[SZ] = {0};
  generate(a, a + SZ, boundedRand);
  int val = boundedRand();
  int* end = remove_copy_if(a, a + SZ, b,
                            bind2nd(greater<int>(), val));
  // Sort for easier viewing:
  sort(a, a + SZ);
  sort(b, end);
  ostream_iterator<int> out(cout, " ");
  cout << "Original Sequence:" << endl;
  copy(a, a + SZ, out); cout << endl;
  cout << "Values <= " << val << endl;
  copy(b, end, out); cout << endl;
} ///:~
```

7.2.5. Adaptadores de puntero a función

```
//: C06:PtrFun1.cpp
// Using ptr_fun() with a unary function.
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int d[] = { 123, 94, 10, 314, 315 };
const int DSZ = sizeof d / sizeof *d;

bool isEven(int x) { return x % 2 == 0; }

int main() {
  vector<bool> vb;
  transform(d, d + DSZ, back_inserter(vb),
```

```
    not1(ptr_fun(isEven)));
  copy(vb.begin(), vb.end(),
    ostream_iterator<bool>(cout, " "));
  cout << endl;
  // Output: 1 0 0 0 1
} ///:~
```

```
//: C06:PtrFun2.cpp {-edg}
// Using ptr_fun() for a binary function.
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

double d[] = { 01.23, 91.370, 56.661,
  023.230, 19.959, 1.0, 3.14159 };
const int DSZ = sizeof d / sizeof *d;

int main() {
  vector<double> vd;
  transform(d, d + DSZ, back_inserter(vd),
    bind2nd(ptr_fun<double, double, double>(pow), 2.0));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, " "));
  cout << endl;
} ///:~
```

```
//: C06:MemFun1.cpp
// Applying pointers to member functions.
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  virtual void draw() { cout << "Circle::Draw()" << endl; }
  ~Circle() { cout << "Circle::~Circle()" << endl; }
};

class Square : public Shape {
public:
  virtual void draw() { cout << "Square::Draw()" << endl; }
```

```
  ~Square() { cout << "Square::~Square()" << endl; }
};

int main() {
  vector<Shape*> vs;
  vs.push_back(new Circle);
  vs.push_back(new Square);
  for_each(vs.begin(), vs.end(), mem_fun(&Shape::draw));
  purge(vs);
} ///:~
```

```
//: C06:MemFun2.cpp
// Calling member functions through an object reference.
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

class Angle {
  int degrees;
public:
  Angle(int deg) : degrees(deg) {}
  int mul(int times) { return degrees *= times; }
};

int main() {
  vector<Angle> va;
  for(int i = 0; i < 50; i += 10)
    va.push_back(Angle(i));
  int x[] = { 1, 2, 3, 4, 5 };
  transform(va.begin(), va.end(), x,
    ostream_iterator<int>(cout, " "),
    mem_fun_ref(&Angle::mul));
  cout << endl;
  // Output: 0 20 60 120 200
} ///:~
```

```
//: C06:FindBlanks.cpp
// Demonstrates mem_fun_ref() with string::empty().
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <fstream>
#include <functional>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

typedef vector<string>::iterator LSI;

int main(int argc, char* argv[]) {
```

```
  char* fname = "FindBlanks.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> vs;
  string s;
  while(getline(in, s))
    vs.push_back(s);
  vector<string> cpy = vs; // For testing
  LSI lsi = find_if(vs.begin(), vs.end(),
     mem_fun_ref(&string::empty));
  while(lsi != vs.end()) {
    *lsi = "A BLANK LINE";
    lsi = find_if(vs.begin(), vs.end(),
      mem_fun_ref(&string::empty));
  }
  for(size_t i = 0; i < cpy.size(); i++)
    if(cpy[i].size() == 0)
      assert(vs[i] == "A BLANK LINE");
    else
      assert(vs[i] != "A BLANK LINE");
} ///:~
```

7.2.6. Escribir sus propios adaptadores de objeto-función

```
//: C06:NumStringGen.h
// A random number generator that produces
// strings representing floating-point numbers.
#ifndef NUMSTRINGGEN_H
#define NUMSTRINGGEN_H
#include <cstdlib>
#include <string>

class NumStringGen {
  const int sz; // Number of digits to make
public:
  NumStringGen(int ssz = 5) : sz(ssz) {}
  std::string operator()() {
    std::string digits("0123456789");
    const int ndigits = digits.size();
    std::string r(sz, ' ');
    // Don't want a zero as the first digit
    r[0] = digits[std::rand() % (ndigits - 1)] + 1;
    // Now assign the rest
    for(int i = 1; i < sz; ++i)
      if(sz >= 3 && i == sz/2)
        r[i] = '.'; // Insert a decimal point
      else
        r[i] = digits[std::rand() % ndigits];
    return r;
  }
};
#endif // NUMSTRINGGEN_H ///:~
```

```
//: C06:MemFun3.cpp
// Using mem_fun().
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

int main() {
  const int SZ = 9;
  vector<string> vs(SZ);
  // Fill it with random number strings:
  srand(time(0)); // Randomize
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
  cout << endl;
  const char* vcp[SZ];
  transform(vs.begin(), vs.end(), vcp,
    mem_fun_ref(&string::c_str));
  vector<double> vd;
  transform(vcp, vcp + SZ, back_inserter(vd),
    std::atof);
  cout.precision(4);
  cout.setf(ios::showpoint);
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

```
//: C06:ComposeTry.cpp
// A first attempt at implementing function composition.
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <string>
using namespace std;

template<typename R, typename E, typename F1, typename F2>
class unary_composer {
  F1 f1;
  F2 f2;
public:
  unary_composer(F1 fone, F2 ftwo) : f1(fone), f2(ftwo) {}
  R operator()(E x) { return f1(f2(x)); }
};

template<typename R, typename E, typename F1, typename F2>
unary_composer<R, E, F1, F2> compose(F1 f1, F2 f2) {
```

```
  return unary_composer<R, E, F1, F2>(f1, f2);
}

int main() {
  double x = compose<double, const string&>(
    atof, mem_fun_ref(&string::c_str))("12.34");
  assert(x == 12.34);
} ///:~
```

```
//: C06:ComposeFinal.cpp {-edg}
// An adaptable composer.
#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

template<typename F1, typename F2> class unary_composer
: public unary_function<typename F2::argument_type,
                        typename F1::result_type> {
  F1 f1;
  F2 f2;
public:
  unary_composer(F1 f1, F2 f2) : f1(f1), f2(f2) {}
  typename F1::result_type
  operator()(typename F2::argument_type x) {
    return f1(f2(x));
  }
};

template<typename F1, typename F2>
unary_composer<F1, F2> compose(F1 f1, F2 f2) {
  return unary_composer<F1, F2>(f1, f2);
}

int main() {
  const int SZ = 9;
  vector<string> vs(SZ);
  // Fill it with random number strings:
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
  cout << endl;
  vector<double> vd;
  transform(vs.begin(), vs.end(), back_inserter(vd),
    compose(ptr_fun(atof), mem_fun_ref(&string::c_str)));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

7.3. Un catálogo de algoritmos STL

7.3.1. Herramientas de soporte para la creación de ejemplos

```
//: C06:PrintSequence.h
// Prints the contents of any sequence.
#ifndef PRINTSEQUENCE_H
#define PRINTSEQUENCE_H
#include <algorithm>
#include <iostream>
#include <iterator>

template<typename Iter>
void print(Iter first, Iter last, const char* nm = "",
           const char* sep = "\n",
           std::ostream& os = std::cout) {
  if(nm != 0 && *nm != '\0')
    os << nm << ": " << sep;
  typedef typename
    std::iterator_traits<Iter>::value_type T;
  std::copy(first, last,
            std::ostream_iterator<T>(std::cout, sep));
  os << std::endl;
}
#endif // PRINTSEQUENCE_H ///:~
```

Reordenación estable vs. inestable

```
//: C06:NString.h
// A "numbered string" that keeps track of the
// number of occurrences of the word it contains.
#ifndef NSTRING_H
#define NSTRING_H
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

typedef std::pair<std::string, int> psi;

// Only compare on the first element
bool operator==(const psi& l, const psi& r) {
  return l.first == r.first;
}

class NString {
  std::string s;
  int thisOccurrence;
```

```
  // Keep track of the number of occurrences:
  typedef std::vector<psi> vp;
  typedef vp::iterator vpit;
  static vp words;
  void addString(const std::string& x) {
    psi p(x, 0);
    vpit it = std::find(words.begin(), words.end(), p);
    if(it != words.end())
      thisOccurrence = ++it->second;
    else {
      thisOccurrence = 0;
      words.push_back(p);
    }
  }
public:
  NString() : thisOccurrence(0) {}
  NString(const std::string& x) : s(x) { addString(x); }
  NString(const char* x) : s(x) { addString(x); }
  // Implicit operator= and copy-constructor are OK here.
  friend std::ostream& operator<<(
    std::ostream& os, const NString& ns) {
    return os << ns.s << " [" << ns.thisOccurrence << "]";
  }
  // Need this for sorting. Notice it only
  // compares strings, not occurrences:
  friend bool
  operator<(const NString& l, const NString& r) {
    return l.s < r.s;
  }
  friend
  bool operator==(const NString& l, const NString& r) {
    return l.s == r.s;
  }
  // For sorting with greater<NString>:
  friend bool
  operator>(const NString& l, const NString& r) {
    return l.s > r.s;
  }
  // To get at the string directly:
  operator const std::string&() const { return s; }
};

// Because NString::vp is a template and we are using the
// inclusion model, it must be defined in this header file:
NString::vp NString::words;
#endif // NSTRING_H ///:~
```

7.3.2. Relleno y generación

Ejemplo

```
//: C06:FillGenerateTest.cpp
// Demonstrates "fill" and "generate."
//{L} Generators
#include <vector>
```

```
#include <algorithm>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  vector<string> v1(5);
  fill(v1.begin(), v1.end(), "howdy");
  print(v1.begin(), v1.end(), "v1", " ");
  vector<string> v2;
  fill_n(back_inserter(v2), 7, "bye");
  print(v2.begin(), v2.end(), "v2");
  vector<int> v3(10);
  generate(v3.begin(), v3.end(), SkipGen(4,5));
  print(v3.begin(), v3.end(), "v3", " ");
  vector<int> v4;
  generate_n(back_inserter(v4),15, URandGen(30));
  print(v4.begin(), v4.end(), "v4", " ");
} ///:~
```

7.3.3. Conteo

Ejemplo

```
//: C06:Counting.cpp
// The counting algorithms.
//{L} Generators
#include <algorithm>
#include <functional>
#include <iterator>
#include <set>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  vector<char> v;
  generate_n(back_inserter(v), 50, CharGen());
  print(v.begin(), v.end(), "v", "");
  // Create a set of the characters in v:
  set<char> cs(v.begin(), v.end());
  typedef set<char>::iterator sci;
  for(sci it = cs.begin(); it != cs.end(); it++) {
    int n = count(v.begin(), v.end(), *it);
    cout << *it << ": " << n << ", ";
  }
  int lc = count_if(v.begin(), v.end(),
    bind2nd(greater<char>(), 'a'));
  cout << "\nLowercase letters: " << lc << endl;
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "sorted", "");
} ///:~
```

7.3.4. Manipulación de secuencias

Ejemplo

```
//: C06:Manipulations.cpp
// Shows basic manipulations.
//{L} Generators
// NString
#include <vector>
#include <string>
#include <algorithm>
#include "PrintSequence.h"
#include "NString.h"
#include "Generators.h"
using namespace std;

int main() {
  vector<int> v1(10);
  // Simple counting:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1.begin(), v1.end(), "v1", " ");
  vector<int> v2(v1.size());
  copy_backward(v1.begin(), v1.end(), v2.end());
  print(v2.begin(), v2.end(), "copy_backward", " ");
  reverse_copy(v1.begin(), v1.end(), v2.begin());
  print(v2.begin(), v2.end(), "reverse_copy", " ");
  reverse(v1.begin(), v1.end());
  print(v1.begin(), v1.end(), "reverse", " ");
  int half = v1.size() / 2;
  // Ranges must be exactly the same size:
  swap_ranges(v1.begin(), v1.begin() + half,
    v1.begin() + half);
  print(v1.begin(), v1.end(), "swap_ranges", " ");
  // Start with a fresh sequence:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1.begin(), v1.end(), "v1", " ");
  int third = v1.size() / 3;
  for(int i = 0; i < 10; i++) {
    rotate(v1.begin(), v1.begin() + third, v1.end());
    print(v1.begin(), v1.end(), "rotate", " ");
  }
  cout << "Second rotate example:" << endl;
  char c[] = "aabbccddeeffgghhiijj";
  const char CSZ = strlen(c);
  for(int i = 0; i < 10; i++) {
    rotate(c, c + 2, c + CSZ);
    print(c, c + CSZ, "", "");
  }
  cout << "All n! permutations of abcd:" << endl;
  int nf = 4 * 3 * 2 * 1;
  char p[] = "abcd";
  for(int i = 0; i < nf; i++) {
    next_permutation(p, p + 4);
    print(p, p + 4, "", "");
```

```
  }
  cout << "Using prev_permutation:" << endl;
  for(int i = 0; i < nf; i++) {
    prev_permutation(p, p + 4);
    print(p, p + 4, "", "");
  }
  cout << "random_shuffling a word:" << endl;
  string s("hello");
  cout << s << endl;
  for(int i = 0; i < 5; i++) {
    random_shuffle(s.begin(), s.end());
    cout << s << endl;
  }
  NString sa[] = { "a", "b", "c", "d", "a", "b",
    "c", "d", "a", "b", "c", "d", "a", "b", "c"};
  const int SASZ = sizeof sa / sizeof *sa;
  vector<NString> ns(sa, sa + SASZ);
  print(ns.begin(), ns.end(), "ns", " ");
  vector<NString>::iterator it =
    partition(ns.begin(), ns.end(),
      bind2nd(greater<NString>(), "b"));
  cout << "Partition point: " << *it << endl;
  print(ns.begin(), ns.end(), "", " ");
  // Reload vector:
  copy(sa, sa + SASZ, ns.begin());
  it = stable_partition(ns.begin(), ns.end(),
    bind2nd(greater<NString>(), "b"));
  cout << "Stable partition" << endl;
  cout << "Partition point: " << *it << endl;
  print(ns.begin(), ns.end(), "", " ");
} ///:~
```

7.3.5. Búsqueda y reemplazo

Ejemplo

```
//: C06:SearchReplace.cpp
// The STL search and replace algorithms.
#include <algorithm>
#include <functional>
#include <vector>
#include "PrintSequence.h"
using namespace std;

struct PlusOne {
  bool operator()(int i, int j) { return j == i + 1; }
};

class MulMoreThan {
  int value;
public:
  MulMoreThan(int val) : value(val) {}
  bool operator()(int v, int m) { return v * m > value; }
};
```

```
int main() {
  int a[] = { 1, 2, 3, 4, 5, 6, 6, 7, 7, 7,
    8, 8, 8, 8, 11, 11, 11, 11, 11 };
  const int ASZ = sizeof a / sizeof *a;
  vector<int> v(a, a + ASZ);
  print(v.begin(), v.end(), "v", " ");
  vector<int>::iterator it = find(v.begin(), v.end(), 4);
  cout << "find: " << *it << endl;
  it = find_if(v.begin(), v.end(),
    bind2nd(greater<int>(), 8));
  cout << "find_if: " << *it << endl;
  it = adjacent_find(v.begin(), v.end());
  while(it != v.end()) {
    cout << "adjacent_find: " << *it
         << ", " << *(it + 1) << endl;
    it = adjacent_find(it + 1, v.end());
  }
  it = adjacent_find(v.begin(), v.end(), PlusOne());
  while(it != v.end()) {
    cout << "adjacent_find PlusOne: " << *it
         << ", " << *(it + 1) << endl;
    it = adjacent_find(it + 1, v.end(), PlusOne());
  }
  int b[] = { 8, 11 };
  const int BSZ = sizeof b / sizeof *b;
  print(b, b + BSZ, "b", " ");
  it = find_first_of(v.begin(), v.end(), b, b + BSZ);
  print(it, it + BSZ, "find_first_of", " ");
  it = find_first_of(v.begin(), v.end(),
    b, b + BSZ, PlusOne());
  print(it,it + BSZ,"find_first_of PlusOne"," ");
  it = search(v.begin(), v.end(), b, b + BSZ);
  print(it, it + BSZ, "search", " ");
  int c[] = { 5, 6, 7 };
  const int CSZ = sizeof c / sizeof *c;
  print(c, c + CSZ, "c", " ");
  it = search(v.begin(), v.end(), c, c + CSZ, PlusOne());
  print(it, it + CSZ,"search PlusOne", " ");
  int d[] = { 11, 11, 11 };
  const int DSZ = sizeof d / sizeof *d;
  print(d, d + DSZ, "d", " ");
  it = find_end(v.begin(), v.end(), d, d + DSZ);
  print(it, v.end(),"find_end", " ");
  int e[] = { 9, 9 };
  print(e, e + 2, "e", " ");
  it = find_end(v.begin(), v.end(), e, e + 2, PlusOne());
  print(it, v.end(),"find_end PlusOne"," ");
  it = search_n(v.begin(), v.end(), 3, 7);
  print(it, it + 3, "search_n 3, 7", " ");
  it = search_n(v.begin(), v.end(),
    6, 15, MulMoreThan(100));
  print(it, it + 6,
    "search_n 6, 15, MulMoreThan(100)", " ");
  cout << "min_element: "
       << *min_element(v.begin(), v.end()) << endl;
  cout << "max_element: "
       << *max_element(v.begin(), v.end()) << endl;
  vector<int> v2;
```

```
  replace_copy(v.begin(), v.end(),
    back_inserter(v2), 8, 47);
  print(v2.begin(), v2.end(), "replace_copy 8 -> 47", " ");
  replace_if(v.begin(), v.end(),
    bind2nd(greater_equal<int>(), 7), -1);
  print(v.begin(), v.end(), "replace_if >= 7 -> -1", " ");
} ///:~
```

7.3.6. Comparación de rangos

Ejemplo

```
//: C06:Comparison.cpp
// The STL range comparison algorithms.
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include "PrintSequence.h"
using namespace std;

int main() {
  // Strings provide a convenient way to create
  // ranges of characters, but you should
  // normally look for native string operations:
  string s1("This is a test");
  string s2("This is a Test");
  cout << "s1: " << s1 << endl << "s2: " << s2 << endl;
  cout << "compare s1 & s1: "
       << equal(s1.begin(), s1.end(), s1.begin()) << endl;
  cout << "compare s1 & s2: "
       << equal(s1.begin(), s1.end(), s2.begin()) << endl;
  cout << "lexicographical_compare s1 & s1: "
       << lexicographical_compare(s1.begin(), s1.end(),
          s1.begin(), s1.end()) <<  endl;
  cout << "lexicographical_compare s1 & s2: "
       << lexicographical_compare(s1.begin(), s1.end(),
          s2.begin(), s2.end()) << endl;
  cout << "lexicographical_compare s2 & s1: "
       << lexicographical_compare(s2.begin(), s2.end(),
          s1.begin(), s1.end()) << endl;
  cout << "lexicographical_compare shortened "
          "s1 & full-length s2: " << endl;
  string s3(s1);
  while(s3.length() != 0) {
    bool result = lexicographical_compare(
      s3.begin(), s3.end(), s2.begin(),s2.end());
    cout << s3 << endl << s2 << ", result = "
         << result << endl;
    if(result == true) break;
    s3 = s3.substr(0, s3.length() - 1);
  }
  pair<string::iterator, string::iterator> p =
    mismatch(s1.begin(), s1.end(), s2.begin());
  print(p.first, s1.end(), "p.first", "");
```

```
  print(p.second, s2.end(), "p.second","");
} ///:~
```

7.3.7. Eliminación de elementos

Ejemplo

```
//: C06:Removing.cpp
// The removing algorithms.
//{L} Generators
#include <algorithm>
#include <cctype>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

struct IsUpper {
  bool operator()(char c) { return isupper(c); }
};

int main() {
  string v;
  v.resize(25);
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v original", "");
  // Create a set of the characters in v:
  string us(v.begin(), v.end());
  sort(us.begin(), us.end());
  string::iterator it = us.begin(), cit = v.end(),
    uend = unique(us.begin(), us.end());
  // Step through and remove everything:
  while(it != uend) {
    cit = remove(v.begin(), cit, *it);
    print(v.begin(), v.end(), "Complete v", "");
    print(v.begin(), cit, "Pseudo v ", " ");
    cout << "Removed element:\t" << *it
         << "\nPsuedo Last Element:\t"
         << *cit << endl << endl;
    ++it;
  }
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v", "");
  cit = remove_if(v.begin(), v.end(), IsUpper());
  print(v.begin(), cit, "v after remove_if IsUpper", " ");
  // Copying versions are not shown for remove()
  // and remove_if().
  sort(v.begin(), cit);
  print(v.begin(), cit, "sorted", " ");
  string v2;
  v2.resize(cit - v.begin());
  unique_copy(v.begin(), cit, v2.begin());
  print(v2.begin(), v2.end(), "unique_copy", " ");
  // Same behavior:
  cit = unique(v.begin(), cit, equal_to<char>());
```

```
  print(v.begin(), cit, "unique equal_to<char>", " ");
} ///:~
```

7.3.8. Ordenación y operación sobre rangos ordenados

Ordenación

Ejemplo

```
//: C06:SortedSearchTest.cpp
// Test searching in sorted ranges.
// NString
#include <algorithm>
#include <cassert>
#include <ctime>
#include <cstdlib>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <iterator>
#include <vector>
#include "NString.h"
#include "PrintSequence.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  typedef vector<NString>::iterator sit;
  char* fname = "Test.txt";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  srand(time(0));
  cout.setf(ios::boolalpha);
  vector<NString> original;
  copy(istream_iterator<string>(in),
    istream_iterator<string>(), back_inserter(original));
  require(original.size() >= 4, "Must have four elements");
  vector<NString> v(original.begin(), original.end()),
    w(original.size() / 2);
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "sort");
  v = original;
  stable_sort(v.begin(), v.end());
  print(v.begin(), v.end(), "stable_sort");
  v = original;
  sit it = v.begin(), it2;
  // Move iterator to middle
  for(size_t i = 0; i < v.size() / 2; i++)
    ++it;
  partial_sort(v.begin(), it, v.end());
  cout << "middle = " << *it << endl;
  print(v.begin(), v.end(), "partial_sort");
  v = original;
  // Move iterator to a quarter position
```

```
  it = v.begin();
  for(size_t i = 0; i < v.size() / 4; i++)
    ++it;
  // Less elements to copy from than to the destination
  partial_sort_copy(v.begin(), it, w.begin(), w.end());
  print(w.begin(), w.end(), "partial_sort_copy");
  // Not enough room in destination
  partial_sort_copy(v.begin(), v.end(), w.begin(),w.end());
  print(w.begin(), w.end(), "w partial_sort_copy");
  // v remains the same through all this process
  assert(v == original);
  nth_element(v.begin(), it, v.end());
  cout << "The nth_element = " << *it << endl;
  print(v.begin(), v.end(), "nth_element");
  string f = original[rand() % original.size()];
  cout << "binary search: "
       << binary_search(v.begin(), v.end(), f) << endl;
  sort(v.begin(), v.end());
  it = lower_bound(v.begin(), v.end(), f);
  it2 = upper_bound(v.begin(), v.end(), f);
  print(it, it2, "found range");
  pair<sit, sit> ip = equal_range(v.begin(), v.end(), f);
  print(ip.first, ip.second, "equal_range");
} ///:~
```

Mezcla de rangos ordenados

Ejemplo

```
//: C06:MergeTest.cpp
// Test merging in sorted ranges.
//{L} Generators
#include <algorithm>
#include "PrintSequence.h"
#include "Generators.h"
using namespace std;

int main() {
  const int SZ = 15;
  int a[SZ*2] = {0};
  // Both ranges go in the same array:
  generate(a, a + SZ, SkipGen(0, 2));
  a[3] = 4;
  a[4] = 4;
  generate(a + SZ, a + SZ*2, SkipGen(1, 3));
  print(a, a + SZ, "range1", " ");
  print(a + SZ, a + SZ*2, "range2", " ");
  int b[SZ*2] = {0}; // Initialize all to zero
  merge(a, a + SZ, a + SZ, a + SZ*2, b);
  print(b, b + SZ*2, "merge", " ");
  // Reset b
  for(int i = 0; i < SZ*2; i++)
    b[i] = 0;
  inplace_merge(a, a + SZ, a + SZ*2);
  print(a, a + SZ*2, "inplace_merge", " ");
  int* end = set_union(a, a + SZ, a + SZ, a + SZ*2, b);
```

```
  print(b, end, "set_union", " ");
} ///:~
```

Ejemplo

```
//: C06:SetOperations.cpp
// Set operations on sorted ranges.
//{L} Generators
#include <algorithm>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  const int SZ = 30;
  char v[SZ + 1], v2[SZ + 1];
  CharGen g;
  generate(v, v + SZ, g);
  generate(v2, v2 + SZ, g);
  sort(v, v + SZ);
  sort(v2, v2 + SZ);
  print(v, v + SZ, "v", "");
  print(v2, v2 + SZ, "v2", "");
  bool b = includes(v, v + SZ, v + SZ/2, v + SZ);
  cout.setf(ios::boolalpha);
  cout << "includes: " << b << endl;
  char v3[SZ*2 + 1], *end;
  end = set_union(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_union", "");
  end = set_intersection(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_intersection", "");
  end = set_difference(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_difference", "");
  end = set_symmetric_difference(v, v + SZ,
    v2, v2 + SZ, v3);
  print(v3, end, "set_symmetric_difference","");
} ///:~
```

7.3.9. Operaciones sobre el montículo

7.3.10. Aplicando una operación a cada elemento de un rango

Ejemplos

```
//: C06:Counted.h
// An object that keeps track of itself.
#ifndef COUNTED_H
#define COUNTED_H
#include <vector>
```

```
#include <iostream>

class Counted {
  static int count;
  char* ident;
public:
  Counted(char* id) : ident(id) { ++count; }
  ~Counted() {
    std::cout << ident << " count = "
              << --count << std::endl;
  }
};

class CountedVector : public std::vector<Counted*> {
public:
  CountedVector(char* id) {
    for(int i = 0; i < 5; i++)
      push_back(new Counted(id));
  }
};
#endif // COUNTED_H ///:~
```

```
//: C06:ForEach.cpp {-mwcc}
// Use of STL for_each() algorithm.
//{L} Counted
#include <algorithm>
#include <iostream>
#include "Counted.h"
using namespace std;

// Function object:
template<class T> class DeleteT {
public:
  void operator()(T* x) { delete x; }
};

// Template function:
template<class T> void wipe(T* x) { delete x; }

int main() {
  CountedVector B("two");
  for_each(B.begin(), B.end(), DeleteT<Counted>());
  CountedVector C("three");
  for_each(C.begin(), C.end(), wipe<Counted>);
} ///:~
```

```
//: C06:Transform.cpp {-mwcc}
// Use of STL transform() algorithm.
//{L} Counted
#include <iostream>
#include <vector>
#include <algorithm>
#include "Counted.h"
using namespace std;
```

```
template<class T> T* deleteP(T* x) { delete x; return 0; }

template<class T> struct Deleter {
  T* operator()(T* x) { delete x; return 0; }
};

int main() {
  CountedVector cv("one");
  transform(cv.begin(), cv.end(), cv.begin(),
    deleteP<Counted>);
  CountedVector cv2("two");
  transform(cv2.begin(), cv2.end(), cv2.begin(),
    Deleter<Counted>());
} ///:~
```

```
//: C06:Inventory.h
#ifndef INVENTORY_H
#define INVENTORY_H
#include <iostream>
#include <cstdlib>
using std::rand;

class Inventory {
  char item;
  int quantity;
  int value;
public:
  Inventory(char it, int quant, int val)
  : item(it), quantity(quant), value(val) {}
  // Synthesized operator= & copy-constructor OK
  char getItem() const { return item; }
  int getQuantity() const { return quantity; }
  void setQuantity(int q) { quantity = q; }
  int getValue() const { return value; }
  void setValue(int val) { value = val; }
  friend std::ostream& operator<<(
    std::ostream& os, const Inventory& inv) {
    return os << inv.item << ": "
      << "quantity " << inv.quantity
      << ", value " << inv.value;
  }
};

// A generator:
struct InvenGen {
  Inventory operator()() {
    static char c = 'a';
    int q = rand() % 100;
    int v = rand() % 500;
    return Inventory(c++, q, v);
  }
};
#endif // INVENTORY_H ///:~
```

```
//: C06:CalcInventory.cpp
// More use of for_each().
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

// To calculate inventory totals:
class InvAccum {
  int quantity;
  int value;
public:
  InvAccum() : quantity(0), value(0) {}
  void operator()(const Inventory& inv) {
    quantity += inv.getQuantity();
    value += inv.getQuantity() * inv.getValue();
  }
  friend ostream&
  operator<<(ostream& os, const InvAccum& ia) {
    return os << "total quantity: " << ia.quantity
              << ", total value: " << ia.value;
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));  // Randomize
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  InvAccum ia = for_each(vi.begin(),vi.end(), InvAccum());
  cout << ia << endl;
} ///:~
```

```
//: C06:TransformNames.cpp
// More use of transform().
#include <algorithm>
#include <cctype>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct NewImproved {
  Inventory operator()(const Inventory& inv) {
    return Inventory(toupper(inv.getItem()),
      inv.getQuantity(), inv.getValue());
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));  // Randomize
```

```
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  transform(vi.begin(),vi.end(),vi.begin(),NewImproved());
  print(vi.begin(), vi.end(), "vi");
} ///:~
```

```
//: C06:SpecialList.cpp
// Using the second version of transform().
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct Discounter {
  Inventory operator()(const Inventory& inv,
    float discount) {
    return Inventory(inv.getItem(), inv.getQuantity(),
      int(inv.getValue() * (1 - discount)));
  }
};

struct DiscGen {
  float operator()() {
    float r = float(rand() % 10);
    return r / 100.0;
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));  // Randomize
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  vector<float> disc;
  generate_n(back_inserter(disc), 15, DiscGen());
  print(disc.begin(), disc.end(), "Discounts:");
  vector<Inventory> discounted;
  transform(vi.begin(),vi.end(), disc.begin(),
    back_inserter(discounted), Discounter());
  print(discounted.begin(), discounted.end(),"discounted");
} ///:~
```

7.3.11. Algoritmos numéricos

Ejemplo

```
//: C06:NumericTest.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
```

```
#include <functional>
#include <numeric>
#include "PrintSequence.h"
using namespace std;

int main() {
  int a[] = { 1, 1, 2, 2, 3, 5, 7, 9, 11, 13 };
  const int ASZ = sizeof a / sizeof a[0];
  print(a, a + ASZ, "a", " ");
  int r = accumulate(a, a + ASZ, 0);
  cout << "accumulate 1: " << r << endl;
  // Should produce the same result:
  r = accumulate(a, a + ASZ, 0, plus<int>());
  cout << "accumulate 2: " << r << endl;
  int b[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2 };
  print(b, b + sizeof b / sizeof b[0], "b", " ");
  r = inner_product(a, a + ASZ, b, 0);
  cout << "inner_product 1: " << r << endl;
  // Should produce the same result:
  r = inner_product(a, a + ASZ, b, 0,
    plus<int>(), multiplies<int>());
  cout << "inner_product 2: " << r << endl;
  int* it = partial_sum(a, a + ASZ, b);
  print(b, it, "partial_sum 1", " ");
  // Should produce the same result:
  it = partial_sum(a, a + ASZ, b, plus<int>());
  print(b, it, "partial_sum 2", " ");
  it = adjacent_difference(a, a + ASZ, b);
  print(b, it, "adjacent_difference 1"," ");
  // Should produce the same result:
  it = adjacent_difference(a, a + ASZ, b, minus<int>());
  print(b, it, "adjacent_difference 2"," ");
} ///:~
```

7.3.12. Utilidades generales

7.4. Creando sus propios algoritmos tipo STL

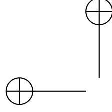
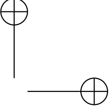
```
// Assumes pred is the incoming condition
replace_copy_if(begin, end, not1(pred));
```

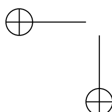
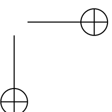
```
//: C06:copy_if.h
// Create your own STL-style algorithm.
#ifndef COPY_IF_H
#define COPY_IF_H

template<typename ForwardIter,
  typename OutputIter, typename UnaryPred>
OutputIter copy_if(ForwardIter begin, ForwardIter end,
  OutputIter dest, UnaryPred f) {
  while(begin != end) {
    if(f(*begin))
      *dest++ = *begin;
```

```
    ++begin;
  }
  return dest;
}
#endif // COPY_IF_H ///:~
```

7.5. Resumen

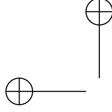
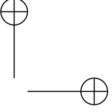
7.6. Ejercicios

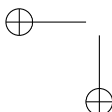
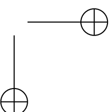



Parte III

Temas especiales

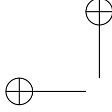
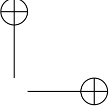
La marca de un profesional aparece en su atención a los detalles más finos del oficio. En esta sección del libro veremos características avanzadas de C++ junto con técnicas de desarrollo usadas por profesionales brillantes de C++.

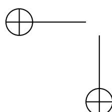
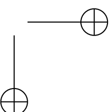
A veces necesita salir de los convencionalismos que suenan a diseño orientado a objetos, inspeccionando el tipo de un objeto en tiempo de ejecución. La mayoría de las veces debería dejar que las funciones virtuales hagan ese trabajo por usted, pero cuando escriba herramientas software para propósitos especiales, tales como depuradores, visualizadores de bases de datos, o navegadores de clases, necesitará determinar la información de tipado en tiempo de ejecución. Ahí es cuando el mecanismo de identificación de tipo en tiempo de ejecución (RTTI) resulta útil. RTTI es el tema del Capítulo 8.

La herencia múltiple ha sido maltratado a lo largo de los años, y algunos lenguajes incluso no la permiten. No obstante, cuando se usa adecuadamente, puede ser una herramienta potente para conseguir código eficiente y elegante. Un buen número de prácticas estándar que involucran herencia múltiple han evolucionado con el tiempo; las veremos en el Capítulo 9.

Quizás la innovación más notable en el desarrollo de software desde las técnicas de orientación a objetos es el uso de los patrones de diseño. Un patrón de diseño describe soluciones para muchos problemas comunes del diseño de software, y se puede aplicar a muchas situaciones e implementación en cualquier lenguaje. En el Capítulo 10 describiremos una selección de patrones de diseño y los implementaremos en C++.

El Capítulo 11 explica los beneficios y desafíos de la programación multihilo. La versión actual de C++ Estándar no especifica soporte para hilos, aunque la mayoría de los sistema operativos los ofrecen. Usaremos un librería portable y disponible libremente para ilustrar cómo los programadores pueden sacar provecho de los hilos para construir aplicaciones más usables y receptivas.

8: Herencia múltiple

El concepto básico de la herencia múltiple (HM) suena bastante simple: puede crear un nuevo tipo heredando de más una una clase base. La sintaxis es exactamente la que espera, y en la medida en que los diagramas de herencia sean simples, la HM puede ser simple también.

Sin embargo, la HM puede presentar un buen número de situaciones ambiguas y extrañas, que se cubren en este capítulo. Pero primero, es útil tener algo de perspectiva sobre el asunto.

8.1. Perspectiva

Antes de C++, el lenguaje orientado a objetos más popular era Smaltalk. Smaltalk fue creado desde cero como un lenguaje orientado a objetos. A menudo se dice que es puro, mientras que a C++ se le llama lenguaje híbrido porque soporta múltiples paradigmas de programación, no sólo el paradigma orientado a objeto. Uno de las decisiones de diseño de Smalltalk fue que todas las clases tendrían solo herencia simple, empezando en una clase base (llamada `Object` - ese es el modelo para la *jerarquía basada en objetos*) [1] En Smalltalk no puede crear una nueva clase sin derivar de un clase existente, que es la razón por la que lleva cierto tiempo ser productivo con Smalltalk: debe aprender la librería de clases antes de empezar a hacer clases nuevas. La jerarquía de clases de Smalltalk es por tanto un único árbol monolítico.

Las clases de Smalltalk normalmente tienen ciertas cosas en común, y siempre tienen algunas cosas en común (las características y el comportamiento de `Objec-t`), de modo que no suelen aparecer situaciones en las que se necesite heredad de más de una clase base. Sin embargo, con C++ puede crear tantos árboles de herencia distintos como quiera. Por completitud lógica el lenguaje debe ser capaz de combinar más de una clase a la vez - por eso la necesidad de herencia múltiple.

No fue obvio, sin embargo, que los programadores requiriesen herencia múltiple, y había (y sigue habiendo) mucha discrepancia sobre si es algo esencial en C++. La HM fue añadida en cfront release 2.0 de AT&T en 1989 y fue el primer cambio significativo en el lenguaje desde la versión 1.0. [2] Desde entonces, se han añadido muchas características al Estándar C++ (las plantillas son dignas de mención) que cambian la manera de pensar al programar y le dan a la HM un papel mucho menos importante. Puede pensar en la HM como una prestación menor del lenguaje que raramente está involucrada en las decisiones de diseño diarias.

Uno de los argumentos más convincentes para la HM involucra a los contenedo-

[1] Esto también ocurre en Java, y en otros lenguajes orientados a objetos.

[2] Son números de versión internos de AT&T.

res. Suponga que quiere crear un contenedor que todo el mundo pueda usar fácilmente. Una propuesta es usar void* como tipo para el contenido. La propuesta de Smalltalk, sin embargo, es hacer un contenedor que aloja `Object`, dado que `Object` es el tipo base de la jerarquía de Smalltalk. Como todo en Smalltalk está derivado de `Object`, un contenedor que aloja `Objects` puede contener cualquier cosa.

Ahora considere la situación en C++. Suponga que el fabricante A crea una jerarquía basada-en-objetos que incluye un conjunto de contenedores incluye uno que desea usar llamado `Holder`. Después, se da cuenta de que la jerarquía de clases del fabricante B contiene alguna clase que también es importante para usted, una clase `BitImage`, por ejemplo, que contiene imágenes. La única forma de hacer que un `Holder` de `BitImage` es derivar de una nueva clase que derive también de `Object`, y así poder almacenarlos en el `Holder`, y `BitImage`:

Figura 8.1:

Éste fue un motivo importante para la HM, y muchas librerías de clases están hechas con este model. Sin embargo, tal como se vio en el Capítulo 5, la aportación de las plantillas ha cambiado la forma de crear contenedores, y por eso esta situación ya no es un asunto crucial en favor de la HM.

El otro motivo por el que se necesita la HM está relacionado con el diseño. Puede usar la HM intencionadamente para hacer un diseño más flexible y útil (o al menos aparentarlo). Un ejemplo de esto es el diseño de la librería original `iostream` (que persiste hoy día, como vio en el Capítulo 4.

Figura 8.2:

Tanto `iostream` como `ostream` son clases útiles por si mismas, pero se pueden derivar simultáneamente por una clase que combina sus características y comportamientos. La clase `ios` proporciona una combinación de las dos clases, y por eso en este caso la HM es un mecanismo de FIXME:code-factoring.

Sin importar lo que le motive a usar HM, debe saber que es más difícil de usar de lo que podría parecer.

8.2. Herencia de interfaces

Un uso no controvertido de la herencia múltiple es la herencia de interfaz. En C++, toda herencia lo es de implementación, dado que todo en una clase base, interface e implentación, pasa a formar parte de la clase derivada. No es posible heredar solo una parte de una clase (es decir, la interface únicamente). Tal como se explica en el [FIXME:enlace en la versión web] Capítulo 14 del volumen 1, es posible hacer herencia privada y protegida para restringir el acceso a los miembros heredados desde las clases base cuando se usa por clientes de instancias de una clase derivada, pero esto no afecta a la propia clase derivada; esa clase sigue conteniendo todos los datos de la clase base y puede acceder a todos los miembros no-privados de la clase base.

La herencia de interfaces. por otra parte, sólo añade declaraciones de miembros a la interfaz de la clase derivada, algo que no está soportado directamente en C++. La técnica habitual para simular la herencia de interfaz en C++ es derivar de una clase interfaz, que es una clase que sólo contiene declaraciones (ni datos ni cuerpos

de funciones). Estas declaraciones serán funciones virtuales puras, excepto el destructor. Aquí hay un ejemplo:

```
//: C09:Interfaces.cpp
// Multiple interface inheritance.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Printable {
public:
  virtual ~Printable() {}
  virtual void print(ostream&) const = 0;
};

class Intable {
public:
  virtual ~Intable() {}
  virtual int toInt() const = 0;
};

class Stringable {
public:
  virtual ~Stringable() {}
  virtual string toString() const = 0;
};

class Able : public Printable, public Intable,
             public Stringable {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const { os << myData; }
  int toInt() const { return myData; }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}

void testIntable(const Intable& n) {
  cout << n.toInt() + 1 << endl;
}

void testStringable(const Stringable& s) {
  cout << s.toString() + "th" << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
```

```
  testIntable(a);
  testStringable(a);
} ///:~
```

La clase `Able` «implementa» las interfaces `Printable`, `Intable` y `Stringable` dado que proporciona implementaciones para las funciones que éstas declaran. Dado que `Able` deriva de las tres clases, los objetos `Able` tienen múltiples relaciones «es-un». Por ejemplo, el objeto `a` puede actuar como un objeto `Printable` dado que su clase, `Able`, deriva públicamente de `Printable` y proporciona una implementación para `print()`. Las funciones de prueba no necesitan saber el tipo más derivado de su parámetro; sólo necesitan un objeto que sea substituible por el tipo de su parámetro.

Como es habitual, una plantilla es una solución más compacta:

```
//: C09:Interfaces2.cpp
// Implicit interface inheritance via templates.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Able {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const { os << myData; }
  int toInt() const { return myData; }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

template<class Printable>
void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}

template<class Intable>
void testIntable(const Intable& n) {
  cout << n.toInt() + 1 << endl;
}

template<class Stringable>
void testStringable(const Stringable& s) {
  cout << s.toString() + "th" << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
  testIntable(a);
  testStringable(a);
```

```
} ///:~
```

Los nombres `Printable`, `Intable` y `Stringable` ahora no son mas que parámetros de la plantilla que asume la existencia de las operaciones indicadas en sus respectivos argumentos. En otras palabras, las funciones de prueba pueden aceptar argumentos de cualquier tipo que proporciona una definición de método con la signatura y tipo de retorno correctos. Hay gente que encuentra más cómoda la primera versión porque los nombres de tipo garantizan que las interfaces esperadas están implementadas. Otros están contentos con el hecho de que si las operaciones requeridas por las funciones de prueba no se satisfacen por los argumentos de la plantilla, el error puede ser capturado en la compilación. Esta segunda es una forma de comprobación de tipos técnicamente más débil que el primer enfoque (herencia), pero el efecto para el programador (y el programa) es el mismo. Se trata de una forma de comprobación débil de tipo que es aceptable para muchos de los programadores C++ de hoy en día.

8.3. Herencia de implementación

```
//: C09:Database.h
// A prototypical resource class.
#ifndef DATABASE_H
#define DATABASE_H
#include <iostream>
#include <stdexcept>
#include <string>

struct DatabaseError : std::runtime_error {
  DatabaseError(const std::string& msg)
    : std::runtime_error(msg) {}
};

class Database {
  std::string dbid;
public:
  Database(const std::string& dbStr) : dbid(dbStr) {}
  virtual ~Database() {}
  void open() throw(DatabaseError) {
    std::cout << "Connected to " << dbid << std::endl;
  }
  void close() {
    std::cout << dbid << " closed" << std::endl;
  }
  // Other database functions...
};
#endif // DATABASE_H ///:~
```

```
//: C09:UseDatabase.cpp
#include "Database.h"

int main() {
  Database db("MyDatabase");
```

```
  db.open();
  // Use other db functions...
  db.close();
}
/* Output:
connected to MyDatabase
MyDatabase closed
*/ ///:~
```

```
//: C09:Countable.h
// A "mixin" class.
#ifndef COUNTABLE_H
#define COUNTABLE_H
#include <cassert>

class Countable {
  long count;
protected:
  Countable() { count = 0; }
  virtual ~Countable() { assert(count == 0); }
public:
  long attach() { return ++count; }
  long detach() {
    return (--count > 0) ? count : (delete this, 0);
  }
  long refCount() const { return count; }
};
#endif // COUNTABLE_H ///:~
```

```
//: C09:DBConnection.h
// Uses a "mixin" class.
#ifndef DBCONNECTION_H
#define DBCONNECTION_H
#include <cassert>
#include <string>
#include "Countable.h"
#include "Database.h"
using std::string;

class DBConnection : public Database, public Countable {
  DBConnection(const DBConnection&); // Disallow copy
  DBConnection& operator=(const DBConnection&);
protected:
  DBConnection(const string& dbStr) throw(DatabaseError)
  : Database(dbStr) { open(); }
  ~DBConnection() { close(); }
public:
  static DBConnection*
  create(const string& dbStr) throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
```

```
  // Other added functionality as desired...
};
#endif // DBCONNECTION_H ///:~
```

```
//: C09:UseDatabase2.cpp
// Tests the Countable "mixin" class.
#include <cassert>
#include "DBConnection.h"

class DBClient {
  DBConnection* db;
public:
  DBClient(DBConnection* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() { db->detach(); }
  // Other database requests using db...
};

int main() {
  DBConnection* db = DBConnection::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
  // Use database, then release attach from original create
  db->detach();
  assert(db->refCount() == 2);
} ///:~
```

```
//: C09:DBConnection2.h
// A parameterized mixin.
#ifndef DBCONNECTION2_H
#define DBCONNECTION2_H
#include <cassert>
#include <string>
#include "Database.h"
using std::string;

template<class Counter>
class DBConnection : public Database, public Counter {
  DBConnection(const DBConnection&); // Disallow copy
  DBConnection& operator=(const DBConnection&);
protected:
  DBConnection(const string& dbStr) throw(DatabaseError)
  : Database(dbStr) { open(); }
  ~DBConnection() { close(); }
public:
  static DBConnection* create(const string& dbStr)
  throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
```

```
    assert(con->refCount() == 1);
    return con;
  }
  // Other added functionality as desired...
};
#endif // DBCONNECTION2_H ///:~
```

```
//: C09:UseDatabase3.cpp
// Tests a parameterized "mixin" class.
#include <cassert>
#include "Countable.h"
#include "DBConnection2.h"

class DBClient {
  DBConnection<Countable>* db;
public:
  DBClient(DBConnection<Countable>* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() { db->detach(); }
};

int main() {
  DBConnection<Countable>* db =
    DBConnection<Countable>::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
  db->detach();
  assert(db->refCount() == 2);
} ///:~
```

```
template<class Mixin1, class Mixin2, ?? , class MixinK>
class Subject : public Mixin1,
                public Mixin2,
                ??
                public MixinK {??};
```

8.4. Subobjetos duplicados

```
//: C09:Offset.cpp
// Illustrates layout of subobjects with MI.
#include <iostream>
using namespace std;

class A { int x; };
class B { int y; };
class C : public A, public B { int z; };
```

```
int main() {
  cout << "sizeof(A) == " << sizeof(A) << endl;
  cout << "sizeof(B) == " << sizeof(B) << endl;
  cout << "sizeof(C) == " << sizeof(C) << endl;
  C c;
  cout << "&c == " << &c << endl;
  A* ap = &c;
  B* bp = &c;
  cout << "ap == " << static_cast<void*>(ap) << endl;
  cout << "bp == " << static_cast<void*>(bp) << endl;
  C* cp = static_cast<C*>(bp);
  cout << "cp == " << static_cast<void*>(cp) << endl;
  cout << "bp == cp? " << boolalpha << (bp == cp) << endl;
  cp = 0;
  bp = cp;
  cout << bp << endl;
}
/* Output:
sizeof(A) == 4
sizeof(B) == 4
sizeof(C) == 12
&c == 1245052
ap == 1245052
bp == 1245056
cp == 1245052
bp == cp? true
0
*/ ///:~
```

```
//: C09:Duplicate.cpp
// Shows duplicate subobjects.
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
};

class Left : public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
  int w;
public:
```

```
  Bottom(int i, int j, int k, int m)
  : Left(i, k), Right(j, k) { w = m; }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl; // 20
} ///:~
```

8.5. Clases base virtuales

```
//: C09:VirtualBase.cpp
// Shows a shared subobject via a virtual base.
#include <iostream>
using namespace std;

class Top {
protected:
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top() {}
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
protected:
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : virtual public Top {
protected:
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    return os << b.x << ',' << b.y << ',' << b.z
      << ',' << b.w;
  }
};
```

```
int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl;
  cout << b << endl;
  cout << static_cast<void*>(&b) << endl;
  Top* p = static_cast<Top*>(&b);
  cout << *p << endl;
  cout << static_cast<void*>(p) << endl;
  cout << dynamic_cast<void*>(p) << endl;
} ///:~
```

```
36
1,2,3,4
1245032
1
1245060
1245032
```

```
//: C09:VirtualBase2.cpp
// How NOT to implement operator<<.
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top() {}
  friend ostream& operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream& operator<<(ostream& os, const Left& l) {
    return os << static_cast<const Top&>(l) << ',' << l.y;
  }
};

class Right : virtual public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
  friend ostream& operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ',' << r.z;
  }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
```

```
  friend ostream& operator<<(ostream& os, const Bottom& b){
    return os << static_cast<const Left&>(b)
      << ',' << static_cast<const Right&>(b)
      << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,1,3,4
} ///:~
```

```
//: C09:VirtualBase3.cpp
// A correct stream inserter.
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top() {}
  friend ostream& operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
protected:
  void specialPrint(ostream& os) const {
    // Only print Left's part
    os << ','<< y;
  }
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream& operator<<(ostream& os, const Left& l) {
    return os << static_cast<const Top&>(l) << ',' << l.y;
  }
};

class Right : virtual public Top {
  int z;
protected:
  void specialPrint(ostream& os) const {
    // Only print Right's part
    os << ','<< z;
  }
public:
  Right(int m, int n) : Top(m) { z = n; }
  friend ostream& operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ',' << r.z;
  }
};

class Bottom : public Left, public Right {
```

```
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream& operator<<(ostream& os, const Bottom& b){
    os << static_cast<const Top&>(b);
    b.Left::specialPrint(os);
    b.Right::specialPrint(os);
    return os << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,3,4
} ///:~
```

```
//: C09:VirtInit.cpp
// Illustrates initialization order with virtual bases.
#include <iostream>
#include <string>
using namespace std;

class M {
public:
  M(const string& s) { cout << "M " << s << endl; }
};

class A {
  M m;
public:
  A(const string& s) : m("in A") {
    cout << "A " << s << endl;
  }
  virtual ~A() {}
};

class B {
  M m;
public:
  B(const string& s) : m("in B")  {
    cout << "B " << s << endl;
  }
  virtual ~B() {}
};

class C {
  M m;
public:
  C(const string& s) : m("in C")  {
    cout << "C " << s << endl;
  }
  virtual ~C() {}
};

class D {
```

```
  M m;
public:
  D(const string& s) : m("in D") {
    cout << "D " << s << endl;
  }
  virtual ~D() {}
};

class E : public A, virtual public B, virtual public C {
  M m;
public:
  E(const string& s) : A("from E"), B("from E"),
  C("from E"), m("in E") {
    cout << "E " << s << endl;
  }
};

class F : virtual public B, virtual public C, public D {
  M m;
public:
  F(const string& s) : B("from F"), C("from F"),
  D("from F"), m("in F") {
    cout << "F " << s << endl;
  }
};

class G : public E, public F {
  M m;
public:
  G(const string& s) : B("from G"), C("from G"),
  E("from G"),  F("from G"), m("in G") {
    cout << "G " << s << endl;
  }
};

int main() {
  G g("from main");
} ///:~
```

```
M in B
B from G
M in C
C from G
M in A
A from E
M in E
E from G
M in D
D from F
M in F
F from G
M in G
G from main
```

8.6. Cuestión sobre búsqueda de nombres

```
//: C09:AmbiguousName.cpp {-xo}
```

```
class Top {
public:
  virtual ~Top() {}
};

class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {
public:
  void f() {}
};

class Bottom : public Left, public Right {};

int main() {
  Bottom b;
  b.f(); // Error here
} ///:~
```

```
//: C09:BreakTie.cpp

class Top {
public:
  virtual ~Top() {}
};

class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {
public:
  void f() {}
};

class Bottom : public Left, public Right {
public:
  using Left::f;
};

int main() {
  Bottom b;
  b.f(); // Calls Left::f()
} ///:~
```

```
//: C09:Dominance.cpp

class Top {
public:
```

```
  virtual ~Top() {}
  virtual void f() {}
};

class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {};

class Bottom : public Left, public Right {};

int main() {
  Bottom b;
  b.f(); // Calls Left::f()
} ///:~
```

```
//: C09:Dominance2.cpp
#include <iostream>
using namespace std;

class A {
public:
  virtual ~A() {}
  virtual void f() { cout << "A::f\n"; }
};

class B : virtual public A {
public:
  void f() { cout << "B::f\n"; }
};

class C : public B {};
class D : public C, virtual public A {};

int main() {
  B* p = new D;
  p->f(); // Calls B::f()
  delete p;
} ///:~
```

8.7. Evitar la MI

8.8. Extender una interface

```
//: C09:Vendor.h
// Vendor-supplied class header
// You only get this & the compiled Vendor.obj.
#ifndef VENDOR_H
#define VENDOR_H
```

```
class Vendor {
public:
  virtual void v() const;
  void f() const; // Might want this to be virtual...
  ~Vendor(); // Oops! Not virtual!
};

class Vendor1 : public Vendor {
public:
  void v() const;
  void f() const;
  ~Vendor1();
};

void A(const Vendor&);
void B(const Vendor&);
// Etc.
#endif // VENDOR_H ///:~
```

```
//: C09:Vendor.cpp {O}
// Assume this is compiled and unavailable to you.
#include "Vendor.h"
#include <iostream>
using namespace std;

void Vendor::v() const { cout << "Vendor::v()" << endl; }

void Vendor::f() const { cout << "Vendor::f()" << endl; }

Vendor::~Vendor() { cout << "~Vendor()" << endl; }

void Vendor1::v() const { cout << "Vendor1::v()" << endl; }

void Vendor1::f() const { cout << "Vendor1::f()" << endl; }

Vendor1::~Vendor1() { cout << "~Vendor1()" << endl; }

void A(const Vendor& v) {
  // ...
  v.v();
  v.f();
  // ...
}

void B(const Vendor& v) {
  // ...
  v.v();
  v.f();
  // ...
} ///:~
```

```
//: C09:Paste.cpp
//{L} Vendor
// Fixing a mess with MI.
```

```
#include <iostream>
#include "Vendor.h"
using namespace std;

class MyBase { // Repair Vendor interface
public:
  virtual void v() const = 0;
  virtual void f() const = 0;
  // New interface function:
  virtual void g() const = 0;
  virtual ~MyBase() { cout << "~MyBase()" << endl; }
};

class Paste1 : public MyBase, public Vendor1 {
public:
  void v() const {
    cout << "Paste1::v()" << endl;
    Vendor1::v();
  }
  void f() const {
    cout << "Paste1::f()" << endl;
    Vendor1::f();
  }
  void g() const { cout << "Paste1::g()" << endl; }
  ~Paste1() { cout << "~Paste1()" << endl; }
};

int main() {
  Paste1& p1p = *new Paste1;
  MyBase& mp = p1p; // Upcast
  cout << "calling f()" << endl;
  mp.f();  // Right behavior
  cout << "calling g()" << endl;
  mp.g(); // New behavior
  cout << "calling A(p1p)" << endl;
  A(p1p); // Same old behavior
  cout << "calling B(p1p)" << endl;
  B(p1p);  // Same old behavior
  cout << "delete mp" << endl;
  // Deleting a reference to a heap object:
  delete &mp; // Right behavior
} ///:~
```

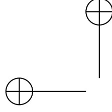
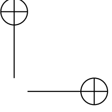
```
MyBase* mp = p1p; // Upcast
```

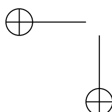
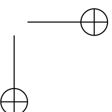
```
calling f()
Paste1::f()
Vendor1::f()
calling g()
Paste1::g()
calling A(p1p)
Paste1::v()
Vendor1::v()
Vendor::f()
calling B(p1p)
Paste1::v()
Vendor1::v()
Vendor::f()
```

```
delete mp
~Paste1()
~Vendor1()
~Vendor()
~MyBase()
```

8.9. Resumen

8.10. Ejercicios

9: Patrones de Diseño

"...describa un problema que sucede una y otra vez en nuestro entorno, y luego describa el núcleo de la solución a ese problema, de tal forma que pueda utilizar esa solución un millón de veces más, sin siquiera hacerlo dos veces de la misma manera." - Christopher Alexander

Este capítulo presenta el importante y aún no tradicional enfoque de los patrones para el diseño de programas.

El avance reciente más importante en el diseño orientado a objetos es probablemente el movimiento de los patrones de diseño, inicialmente narrado en "Design Patterns", por Gamma, Helm, Johnson y Vlissides (Addison Wesley, 1995), que suele llamarse el libro de la "Banda de los Cuatro" (en inglés, GoF: Gang of Four). El GoF muestra 23 soluciones para clases de problemas muy particulares. En este capítulo se discuten los conceptos básicos de los patrones de diseño y se ofrecen ejemplos de código que ilustran los patrones escogidos. Esto debería abrirle el apetito para leer más acerca de los patrones de diseño, una fuente de lo que se ha convertido en vocabulario esencial, casi obligatorio, para la programación orientada a objetos.

9.1. El Concepto de Patrón

En principio, puede pensar en un patrón como una manera especialmente inteligente e intuitiva de resolver una clase de problema en particular. Parece que un equipo de personas han estudiado todos los ángulos de un problema y han dado con la solución más general y flexible para ese tipo de problema. Este problema podría ser uno que usted ha visto y resuelto antes, pero su solución probablemente no tenía la clase de completitud que verá plasmada en un patrón. Es más, el patrón existe independientemente de cualquier implementación particular y puede implementarse de numerosas maneras.

Aunque se llaman "patrones de diseño", en realidad no están ligados al ámbito del diseño. Un patrón parece apartarse de la manera tradicional de pensar sobre el análisis, diseño e implementación. En cambio, un patrón abarca una idea completa dentro de un programa, y por lo tanto puede también abarcar las fases de análisis y diseño de alto nivel. Sin embargo, dado que un patrón a menudo tiene una implementación directa en código, podría no mostrarse hasta el diseño de bajo nivel o la implementación (y usted no se daría cuenta de que necesita ese patrón hasta que llegase a esas fases).

El concepto básico de un patrón puede verse también como el concepto básico del diseño de programas en general: añadir capas de abstracción. Cuando se abstrae algo, se están aislando detalles concretos, y una de las razones de mayor peso para hacerlo es separar las cosas que cambian de las cosas que no. Otra forma de verlo es que una vez que encuentra una parte de su programa que es susceptible de cambiar,

querrá prevenir que esos cambios propagen efectos colaterales por su código. Si lo consigue, su código no sólo será más fácil de leer y comprender, también será más fácil de mantener, lo que a la larga, siempre redunda en menores costes.

La parte más difícil de desarrollar un diseño elegante y mantenible a menudo es descubrir lo que llamamos el "vector de cambio". (Aquí "vector" se refiere al mayor gradiente tal y como se entiende en ciencias, no como la clase contenedora.) Esto implica encontrar la cosa más importante que cambia en su sistema o, dicho de otra forma, descubrir dónde están sus mayores costes. Una vez que descubra el vector de cambios, tendrá el punto focal alrededor del cual estructurar su diseño.

Por lo tanto, el objetivo de los patrones de diseño es encapsular el cambio. Si lo enfoca de esta forma, ya habrá visto algunos patrones de diseño en este libro. Por ejemplo, la herencia podría verse como un patrón de diseño (aunque uno implementado por el compilador). Expresa diferencias de comportamiento (eso es lo que cambia) en objetos que tienen todos la misma interfaz (esto es lo que no cambia). La composición también podría considerarse un patrón, ya que puede cambiar dinámica o estáticamente los objetos que implementan su clase, y por lo tanto, la forma en la que funciona la clase. Normalmente, sin embargo, las características que los lenguajes de programación soportan directamente no se han clasificado como patrones de diseño.

También ha visto ya otro patrón que aparece en el GoF: el «iterador». Esta es la herramienta fundamental usada en el diseño del STL, descrito en capítulos anteriores. El iterador esconde la implementación concreta del contenedor a medida que se avanza y se seleccionan los elementos uno a uno. Los iteradores le ayudan a escribir código genérico que realiza una operación en todos los elementos de un rango sin tener en cuenta el contenedor que contiene el rango. Por lo tanto, cualquier contenedor que pueda producir iteradores puede utilizar su código genérico.

9.1.1. La composición es preferible a la herencia

La contribución más importante del GoF puede que no sea un patrón, si no una máxima que introducen en el Capítulo 1: «Prefiera siempre la composición de objetos antes que la herencia de clases». Entender la herecia y el polimorfismo es un reto tal, que podría empezar a otorgarle una importancia excesiva a estas técnicas. Se ven muchos diseños excesivamente complicados (el nuestro incluído) como resultado de ser demasiado indulgentes con la herencia - por ejemplo, muchos diseños de herencia múltiple se desarrollan por insistir en usar la herencia en todas partes.

Una de las directrices en la «Programación Extrema» es "haga la cosa más simple que pueda funcionar". Un diseño que parece requerir de herencia puede a menudo simplificarse drásticamente usando una composición en su lugar, y descubrirá también que el resultado es más flexible, como comprenderá al estudiar algunos de los patrones de diseño de este capítulo. Por lo tanto, al considerar un diseño, pregúntese: ¿Podría ser más simple si usara Composición? ¿De verdad necesito Herencia aquí, y qué me aporta?

9.2. Clasificación de los patrones

El GoF describe 23 patrones, clasificados según tres propósitos FIXME: (all of which revolve around the particular aspect that can vary):

1. Creacional: Cómo se puede crear un objeto. Habitualmente esto incluye aislar los detalles de la creación del objeto, de forma que su código no dependa de los tipos

de objeto que hay y por lo tantok, no tenga que cambiarlo cuando añada un nuevo tipo de objeto. Este capítulo presenta los patrones Singleton, Fábricas (Factories), y Constructor (Builder).

2. Estructural: Esto afecta a la manera en que los objetos se conectan con otros objetos para asegurar que los cambios del sistema no requieren cambiar esas conexiones. Los patrones estructurales suelen imponerlos las restricciones del proyecto. En este capítulo verá el Proxy y el Adaptador (Adapter).

3. Comportacional: Objetos que manejan tipos particulares de acciones dentro de un programa. Éstos encapsulan procesos que quiere que se ejecuten, como interpretar un lenguaje, completar una petición, moverse a través de una secuencia (como en un iterador) o implementar un algoritmo. Este capítulo contiene ejemplos de `Comando` (Command), Método Plantilla (Template Method), Estado (State), Estrategia (Strategy), Cadena de Responsabilidad (Chain of Responsibility), Observador (Observer), FIXME: Despachador Múltiple (Multiple Dispatching) y Visitador (Visitor).

El GoF incluye una sección sobre cada uno de los 23 patrones, junto con uno o más ejemplos de cada uno, típicamente en C++ aunque a veces en SmallTalk. Este libro no repite los detalles de los patrones mostrados en GoF, ya que aquél FIXME: "stands on its own" y debería estudiarse aparte. La descripción y los ejemplos que se dan aquí intentan darle una visión de los patrones, de forma que pueda hacerse una idea de lo que tratan y de porqué son importantes.

9.2.1. Características, modismos patrones

El trabajo va más allá de lo que se muestra en el libro del GoF. Desde su publicación, hay más patrones y un proceso más refinado para definir patrones de diseño.[135] Esto es importante porque no es fácil identificar nuevos patrones ni describirlos adecuadamente. Hay mucha confusión en la literatura popular acerca de qué es un patrón de diseño, por ejemplo. Los patrones no son triviales, ni están representados por características implementadas en un lenguaje de programación. Los constructores y destructores, por ejemplo, podrían llamarse el patrón de inicialización garantizada y el de limpieza. Hay constructores importantes y esenciales, pero son características del lenguaje rutinarias, y no son lo suficientemente ricas como para ser consideradas patrones.

Otro FIXME: (no-ejemplo? anti-ejemplo?) viene de varias formas de agregación. La agregación es un principio completamente fundamental en la programación orientada a objetos: se hacen objetos a partir de otros objetos. Aunque a veces, esta idea se clasifica erróneamente como un patrón. Esto no es bueno, porque contamina la idea del patrón de diseño, y sugiere que cualquier cosa que le sorprenda la primera vez que la ve debería convertirse en un patrón de diseño.

El lenguaje Java da otro ejemplo equivocado: Los diseñadores de la especificación de JavaBeans decidieron referirse a la notación get/set como un patrón de diseño (por ejemplo, getInfo() devuelve una propiedad Info y setInfo() la modifica). Esto es únicamente una convención de nombrado, y de ninguna manera constituye un patrón de diseño.

9.3. Simplificación de modismos

Antes de adentrarnos en técnicas más complejas, es útil echar un vistazo a algunos métodos básicos de mantener el código simple y sencillo.

9.3.1. Mensajero

El más trivial es el Mensajero (Messenger), [136] que empaqueta información en un objeto que se envia, en lugar de ir enviando todas las piezas independientemente. Nótese que sin el Mensajero, el código para la función translate() sería mucho más confuso:

```
//: C10:MessengerDemo.cpp
#include <iostream>
#include <string>
using namespace std;

class Point { // A messenger
public:
  int x, y, z; // Since it's just a carrier
  Point(int xi, int yi, int zi) : x(xi), y(yi), z(zi) {}
  Point(const Point& p) :  x(p.x), y(p.y), z(p.z) {}
  Point& operator=(const Point& rhs) {
    x = rhs.x;
    y = rhs.y;
    z = rhs.z;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const Point& p) {
    return os << "x=" << p.x << " y=" << p.y
              << " z=" << p.z;
  }
};

class Vector { // Mathematical vector
public:
  int magnitude, direction;
  Vector(int m, int d) : magnitude(m), direction(d) {}
};

class Space {
public:
  static Point translate(Point p, Vector v) {
    // Copy-constructor prevents modifying the original.
    // A dummy calculation:
    p.x += v.magnitude + v.direction;
    p.y += v.magnitude + v.direction;
    p.z += v.magnitude + v.direction;
    return p;
  }
};

int main() {
  Point p1(1, 2, 3);
  Point p2 = Space::translate(p1, Vector(11, 47));
  cout << "p1: " << p1 << " p2: " << p2 << endl;
} ///:~
```

El código ha sido simplificado para evitar distracciones.

Como el objetivo del Mensajero es simplemente llevar datos, dichos datos se hacen públicos para facilitar el acceso. Sin embargo, podría tener razones para hacer estos campos privados.

9.3.2. Parámetro de Recolección

El hermano mayor del Mensajero es el parámetro de recolección, cuyo trabajo es capturar información sobre la función a la que es pasado. Generalmente se usa cuando el parámetro de recolección se pasa a múltiples funciones; es como una abeja recogiendo polen.

Un contenedor (container) es un parámetro de recolección especialmente útil, ya que está configurado para añadir objetos dinámicamente:

```
//: C10:CollectingParameterDemo.cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class CollectingParameter : public vector<string> {};

class Filler {
public:
  void f(CollectingParameter& cp) {
    cp.push_back("accumulating");
  }
  void g(CollectingParameter& cp) {
    cp.push_back("items");
  }
  void h(CollectingParameter& cp) {
    cp.push_back("as we go");
  }
};

int main() {
  Filler filler;
  CollectingParameter cp;
  filler.f(cp);
  filler.g(cp);
  filler.h(cp);
  vector<string>::iterator it = cp.begin();
  while(it != cp.end())
    cout << *it++ << " ";
  cout << endl;
} ///:~
```

El parámetro de recolección debe tener alguna forma de establecer o insertar valores. Nótese que por esta definición, un Mensajero podría usarse como parámetro de recolección. La clave reside en que el parámetro de recolección se pasa y es modificado por la función que lo recibe.

9.4. Singleton

Posiblemente, el patrón de diseño más simple del GoF es el `Singleton`, que es una forma de asegurar una única instancia de una clase. El siguiente programa muestra cómo implementar un `Singleton` en C++:

```
//: C10:SingletonPattern.cpp
#include <iostream>
using namespace std;

class Singleton {
  static Singleton s;
  int i;
  Singleton(int x) : i(x) { }
  Singleton& operator=(Singleton&);  // Disallowed
  Singleton(const Singleton&);       // Disallowed
public:
  static Singleton& instance() { return s; }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

Singleton Singleton::s(47);

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

La clave para crear un Singleton es evitar que el programador cliente tenga control sobre el ciclo de vida del objeto. Para lograrlo, declare todos los constructores privados, y evite que el compilador genere implícitamente cualquier constructor. Fíjese que el FIXME: constructor de copia? y el operador de asignación (que intencionadamente carecen de implementación alguna, ya que nunca van a ser llamados) están declarados como privados, para evitar que se haga cualquier tipo de copia.

También debe decidir cómo va a crear el objeto. Aquí, se crea de forma estática, pero también puede esperar a que el programador cliente pida uno y crearlo bajo demanda. Esto se llama "inicialización vaga", y sólo tiene sentido si resulta caro crear el objeto y no siempre se necesita.

Si devuelve un puntero en lugar de una referencia, el usuario podría borrar el puntero sin darse cuenta, por lo que la implementación citada anteriormente es más segura (el destructor también podría declararse privado o protegido para solventar el problema). En cualquier caso, el objeto debería almacenarse de forma privada.

Usted da acceso a través de FIXME (funciones de miembros) públicas. Aquí, `instance()` genera una referencia al objeto `Singleton`. El resto de la interfaz (`getValue()` y `setValue()`) es la interfaz regular de la clase.

Fíjese en que no está restringido a crear un único objeto. Esta técnica también soporta la creacion de un `pool` limitado de objetos. En este caso, sin embargo, puede enfrentarse al problema de compartir objetos del `pool`. Si esto supone un problema,

puede crear una solución que incluya un `check-out` y un `check-in` de los objetos compartidos.

9.4.1. Variantes del Singleton

Cualquier miembro `static` dentro de una clase es una forma de Singleton se hará uno y sólo uno. En cierto modo, el lenguaje da soporte directo a esta idea; se usa de forma regular. Sin embargo, los objetos estáticos tienen un problema (ya miembros o no): el orden de inicialización, tal y como se describe en el volumen 1 de este libro. Si un objeto `static` depende de otro, es importante que los objetos se inicializen en el orden correcto.

En el volumen 1, se mostró cómo controlar el orden de inicialización definiendo un objeto estático dentro de una función. Esto retrasa la inicialización del objeto hasta la primera vez que se llama a la función. Si la función devuelve una referencia al objeto estático, hace las veces de `Singleton` a la vez que elimina gran parte de la preocupación de la inicialización estática. Por ejemplo, suponga que quiere crear un fichero de `log` en la primera llamada a una función que devuelve una referencia a dicho fichero. Basta con este fichero de cabecera:

```
//: C10:LogFile.h
#ifndef LOGFILE_H
#define LOGFILE_H
#include <fstream>
std::ofstream& logfile();
#endif // LOGFILE_H ///:~
```

La implementación no debe FIXME: hacerse en la misma línea, porque eso significaría que la función entera, incluída la definición del objeto estático que contiene, podría ser duplicada en cualquier unidad de traducción donde se incluya, lo que viola la regla de única definición de C++. [137] Con toda seguridad, esto frustraría cualquier intento de controlar el orden de inicialización (pero potencialmente de una forma sutil y difícil de detectar). De forma que la implementación debe separarse:

```
//: C10:LogFile.cpp {O}
#include "LogFile.h"
std::ofstream& logfile() {
  static std::ofstream log("Logfile.log");
  return log;
} ///:~
```

Ahora el objeto `log` no se inicializará hasta la primera vez que se llame a `logfile()`. Así que, si crea una función:

```
//: C10:UseLog1.h
#ifndef USELOG1_H
#define USELOG1_H
void f();
#endif // USELOG1_H ///:~
```

que use `logfile()` en su implementación:

```
//: C10:UseLog1.cpp {O}
#include "UseLog1.h"
#include "LogFile.h"
void f() {
  logfile() << __FILE__ << std::endl;
} ///:~
```

y utiliza `logfile()` otra vez en otro fichero:

```
//: C10:UseLog2.cpp
//{L} LogFile UseLog1
#include "UseLog1.h"
#include "LogFile.h"
using namespace std;
void g() {
  logfile() << __FILE__ << endl;
}

int main() {
  f();
  g();
} ///:~
```

el objecto `log` no se crea hasta la primera llamada a `f()`.

Puede combinar fácilmente la creación de objetos estáticos dentro de una función miembro con la clase `Singleton`. SingletonPattern.cpp puede modificarse para usar esta aproximación:[138]

```
//: C10:SingletonPattern2.cpp
// Meyers' Singleton.
#include <iostream>
using namespace std;

class Singleton {
  int i;
  Singleton(int x) : i(x) { }
  void operator=(Singleton&);
  Singleton(const Singleton&);
public:
  static Singleton& instance() {
    static Singleton s(47);
    return s;
  }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
```

```
} ///:~
```

Se da un caso especialmente interesante cuando dos `Singletons` dependen mutuamente el uno del otro, de esta forma:

```
//: C10:FunctionStaticSingleton.cpp

class Singleton1 {
  Singleton1() {}
public:
  static Singleton1& ref() {
    static Singleton1 single;
    return single;
  }
};

class Singleton2 {
  Singleton1& s1;
  Singleton2(Singleton1& s) : s1(s) {}
public:
  static Singleton2& ref() {
    static Singleton2 single(Singleton1::ref());
    return single;
  }
  Singleton1& f() { return s1; }
};

int main() {
  Singleton1& s1 = Singleton2::ref().f();
} ///:~
```

Cuando se llama a `Singleton2::ref()`, hace que se cree su único objeto `Singleton2`. En el proceso de esta creación, se llama a `Singleton1::ref()`, y esto hace que se cree su objeto único `Singleton1`. Como esta técnica no se basa en el orden de linkado ni el de carga, el programador tiene mucho mayor control sobre la inicialización, lo que redunda en menos problemas.

Otra variación del `Singleton` separa la unicidad de un objeto de su implementación. Esto se logra usando el "Patrón Plantilla Curiosamente Recursivo" mencionado en el Capítulo 5:

```
//: C10:CuriousSingleton.cpp
// Separates a class from its Singleton-ness (almost).
#include <iostream>
using namespace std;

template<class T> class Singleton {
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);
protected:
  Singleton() {}
  virtual ~Singleton() {}
public:
  static T& instance() {
```

```
    static T theInstance;
    return theInstance;
  }
};

// A sample class to be made into a Singleton
class MyClass : public Singleton<MyClass> {
  int x;
protected:
  friend class Singleton<MyClass>;
  MyClass() { x = 0; }
public:
  void setValue(int n) { x = n; }
  int getValue() const { return x; }
};

int main() {
  MyClass& m = MyClass::instance();
  cout << m.getValue() << endl;
  m.setValue(1);
  cout << m.getValue() << endl;
} ///:~
```

`MyClass` se convierte en `Singleton`:

1. Haciendo que su constructor sea `private` o `protected`.

2. Haciéndose amigo de `Singleton<MyClass>`.

3. Derivando `MyClass` desde `Singleton<MyClass>`.

La auto-referencia del paso 3 podría sonar inversímil, pero tal como se explicó en el Capítulo 5, funciona porque sólo hay una dependencia estática sobre el argumento plantilla de la plantilla `Singleton`. En otras palabras, el código de la clase `Singleton<MyClass>` puede ser instanciado por el compilador porque no depende del tamaño de `MyClass`. Es después, cuando se a `Singleton<MyClass>::instance()`, cuando se necesita el tamaño de `MyClass`, y para entonces `MyClass` ya se ha compilado y su tamaño se conoce.[139]

Es interesante lo intrincado que un patrón tan simple como el `Singleton` puede llegar a ser, y ni siquiera se han tratado todavía asuntos de seguridad de hilos. Por último, el patrón `Singleton` debería usarse lo justo y necesario. Los verdaderos objetos `Singleton` rara vez aparecen, y la última cosa para la que debe usarse un `Singleton` es para remplazar a una variable global. [140]

9.5. Comando: elegir la operación

El patrón `Comando` es estructuralmente muy sencillo, pero puede tener un impacto importante en el desacoplamiento (y, por ende, en la limpieza) de su código.

En "Advanced C++: Programming Styles And Idioms (Addison Wesley, 1992)", Jim Coplien acuña el término `functor`, que es un objeto cuyo único propósito es encapsular una función (dado que `functor` tiene su significado en matemáticas, usaremos el término "objeto función", que es más explícito). El quid está en desacoplar la elección de la función que hay que llamar del sitio donde se llama a dicha función.

Este término se menciona en el GoF, pero no se usa. Sin embargo, el concepto de "objeto función" se repite en numerosos patrones del libro.

Un `Comando` es un objeto función en su estado más puro: una función que tiene un objeto. Al envolver una función en un objeto, puede pasarla a otras funciones u objetos como parámetro, para decirles que realicen esta operación concreta mientras llevan a cabo su petición. Se podría decir que un `Comando` es un Mensajero que lleva un comportamiento.

```
//: C10:CommandPattern.cpp
#include <iostream>
#include <vector>
using namespace std;

class Command {
public:
  virtual void execute() = 0;
};

class Hello : public Command {
public:
  void execute() { cout << "Hello "; }
};

class World : public Command {
public:
  void execute() { cout << "World! "; }
};

class IAm : public Command {
public:
  void execute() { cout << "I'm the command pattern!"; }
};

// An object that holds commands:
class Macro {
  vector<Command*> commands;
public:
  void add(Command* c) { commands.push_back(c); }
  void run() {
    vector<Command*>::iterator it = commands.begin();
    while(it != commands.end())
      (*it++)->execute();
  }
};

int main() {
  Macro macro;
  macro.add(new Hello);
  macro.add(new World);
  macro.add(new IAm);
  macro.run();
} ///:~
```

El punto principal del `Comando` es permitirle dar una acción deseada a una función u objeto. En el ejemplo anterior, esto provee una manera de encolar un conjunto

de acciones que se deben ejecutar colectivamente. Aquí, puede crear dinámicamente nuevos comportamientos, algo que puede hacer normalmente escribiendo nuevo código, pero en el ejemplo anterior podría hacerse interpretando un `script` (vea el patrón Intérprete si lo que necesita hacer se vuelve demasiado complicado).

Según el GoF, los `Comandos` son un sustituto orientado a objetos de las `retrollamadas` (`callbacks`). [141] Sin embargo, pensamos que la palabra "retro" es una parte esencial del concepto de retrollamada -una retrollamada retorna al creador de la misma. Por otro lado, un objeto `Comando`, simplemente se crea y se entrega a alguna función u objeto, y no se permanece conectado de por vida al objecto `Comando`.

Un ejemplo habitual del patrón `Comando` es la implementación de la funcionalidad de "deshacer" en una aplicación. Cada vez que el usuario realiza una operación, se coloca el correspondiente objeto Comando de deshacer en una cola. Cada objeto Comando que se ejecuta guarda el estado del programa en el paso anterior.

9.5.1. Desacoplar la gestión de eventos con Comando

Como se verá en el siguiente capítulo, una de las razones para emplear técnicas de concurrencia es facilitar la gestión de la programación dirigida por eventos, donde los eventos pueden aparecer en el programa de forma impredecible. Por ejemplo, un usuario que pulsa un botón de "Salir" mientras se está realizando una operación espera que el programa responda rápidamente.

Un motivo para usar concurrencia es que previene el aclopamiento entre los bloques del código. Es decir, si está ejecutando un hilo aparte para vigilar el botónde salida, las operaciones normales de su programa no necesitan saber nada sobre el botón ni sobe ninguna de las demás operaciones que se están vigilando.

Sin embargo, una vez que comprenda que el quiz está en el acoplamiento, puede evitarlo usando el patrón Comando. Cada operación normal debe llamar periódicamente a una función para que compruebe el estado de los eventos, pero con el patrón Comando, estas operaciones normales no tienen porqué saber nada sobre lo que están comprobando, y por lo tanto, están desacopladas del código de manejo de eventos:

```
//: C10:MulticastCommand.cpp {RunByHand}
// Decoupling event management with the Command pattern.
#include <iostream>
#include <vector>
#include <string>
#include <ctime>
#include <cstdlib>
using namespace std;

// Framework for running tasks:
class Task {
public:
  virtual void operation() = 0;
};

class TaskRunner {
  static vector<Task*> tasks;
  TaskRunner() {} // Make it a Singleton
  TaskRunner& operator=(TaskRunner&); // Disallowed
  TaskRunner(const TaskRunner&); // Disallowed
```

```
  static TaskRunner tr;
public:
  static void add(Task& t) { tasks.push_back(&t); }
  static void run() {
    vector<Task*>::iterator it = tasks.begin();
    while(it != tasks.end())
      (*it++)->operation();
  }
};

TaskRunner TaskRunner::tr;
vector<Task*> TaskRunner::tasks;

class EventSimulator {
  clock_t creation;
  clock_t delay;
public:
  EventSimulator() : creation(clock()) {
    delay = CLOCKS_PER_SEC/4 * (rand() % 20 + 1);
    cout << "delay = " << delay << endl;
  }
  bool fired() {
    return clock() > creation + delay;
  }
};

// Something that can produce asynchronous events:
class Button {
  bool pressed;
  string id;
  EventSimulator e; // For demonstration
public:
  Button(string name) : pressed(false), id(name) {}
  void press() { pressed = true; }
  bool isPressed() {
    if(e.fired()) press(); // Simulate the event
    return pressed;
  }
  friend ostream&
  operator<<(ostream& os, const Button& b) {
    return os << b.id;
  }
};

// The Command object
class CheckButton : public Task {
  Button& button;
  bool handled;
public:
  CheckButton(Button & b) : button(b), handled(false) {}
  void operation() {
    if(button.isPressed() && !handled) {
      cout << button << " pressed" << endl;
      handled = true;
    }
  }
};
```

```
// The procedures that perform the main processing. These
// need to be occasionally "interrupted" in order to
// check the state of the buttons or other events:
void procedure1() {
  // Perform procedure1 operations here.
  // ...
  TaskRunner::run(); // Check all events
}

void procedure2() {
  // Perform procedure2 operations here.
  // ...
  TaskRunner::run(); // Check all events
}

void procedure3() {
  // Perform procedure3 operations here.
  // ...
  TaskRunner::run(); // Check all events
}

int main() {
  srand(time(0)); // Randomize
  Button b1("Button 1"), b2("Button 2"), b3("Button 3");
  CheckButton cb1(b1), cb2(b2), cb3(b3);
  TaskRunner::add(cb1);
  TaskRunner::add(cb2);
  TaskRunner::add(cb3);
  cout << "Control-C to exit" << endl;
  while(true) {
    procedure1();
    procedure2();
    procedure3();
  }
} ///:~
```

Aquí, el objeto Comando está representado por `Tarea`s ejecutadas por el Singleton `TaskRunner`. `EventSimulator` crea un retraso aleatorio, de modo que si se llama periódicamente a la función fired() el resultado cambiará de `false` a `true` en algún momento aleatorio. Los objetos `EventSimulator` se utilizan dentro de los Botones para simular que ocurre un evento de usuario en un momento impredecible. `CheckButton` es la implementación de la Tarea que es comprobada periódicamente por todo el código "normal" del progama. Puede ver cómo ocurre al final de procedure1(), procedure2() y procedure3().

Aunque esto requiere un poco más de razonamiento para establecerlo, verá en el Capítulo 11 que utilizar hilos requiere mucho pensamiento y cuidado para prevenir las muchas dificultades inhenerentes a la programación concurrente, por lo que la solución más simple puede ser preferible. También puede crear un esquema de hilos muy simple moviendo las llamadas a TaskRunner::run() a un objeto temporizador multi-hilo. Al hacer esto, se elimina todo el acoplamiento entre las operaciones "normales" (los "procedures" en el ejemplo anterior) y el código de eventos.

9.6. Desacoplamiento de objetos

Tanto el Proxy como el Estado proporcionen una clase sucedánea. El código habla a esta clase sucedánea, y la verdadera clase que hace el trabajo está escondida detrás de la sucedánea. Cuando usted llama a una función en la clase sucedánea, simplemente da un rodeo y llama a la función en la clase implementadora. Estos dos patrones son tan familiares que, estructuralmente, Proxy es un caso especial de Estado. Uno está tentado de juntar ambos en un patrón llamado Sucedánea, pero la intención de los dos patrones es distinta. Puede ser fácil caer en la trampa de pensar que si la estructura es la misma, los patrones son el mismo. Debe mirar siempre la intención del patrón para tener claro lo que hace.

La idea básica es simple: cree una clase base, la sucedánea se deriva junto con la clase o clases que aportan la siguiente implementación:

Cuando se crea una clase sucedánea, se le da una implementación a la que envía las llamadas a función.

Estructuralmente, la diferencia entre Proxy y Estado es simple: un Proxy sólo tiene una implementación, mientras que Estado tiene más de una. La aplicación de los patrones se considera (en el GoF) distinta: Proxy controla el acceso a su implementación, mientras que Estado cambia la implementación dinámicamente. Sin embargo, si se amplía la noción de "controlar el acceso a la implementación", entonces los dos parecen ser parte de un todo.

9.6.1. Proxy: FIXME: hablando en nombre de otro objeto

Si se implementa un Proxy usando el diagrama anterior, tiene esta pinta:

```
//: C10:ProxyDemo.cpp
// Simple demonstration of the Proxy pattern.
#include <iostream>
using namespace std;

class ProxyBase {
public:
  virtual void f() = 0;
  virtual void g() = 0;
  virtual void h() = 0;
  virtual ~ProxyBase() {}
};

class Implementation : public ProxyBase {
public:
  void f() { cout << "Implementation.f()" << endl; }
  void g() { cout << "Implementation.g()" << endl; }
  void h() { cout << "Implementation.h()" << endl; }
};

class Proxy : public ProxyBase {
  ProxyBase* implementation;
public:
  Proxy() { implementation = new Implementation(); }
  ~Proxy() { delete implementation; }
  // Forward calls to the implementation:
  void f() { implementation->f(); }
  void g() { implementation->g(); }
```

```
  void h() { implementation->h(); }
};

int main()  {
  Proxy p;
  p.f();
  p.g();
  p.h();
} ///:~
```

En algunos casos, Implementation no necesita la misma interfaz que Proxy, siempre y cuando Proxy esté de alguna forma hablando en nombre de la clase Implementación y referenciando llamadas a función hacia ella, entonces la idea básica se satisface (note que esta afirmación está reñida con la definición de Proxy del GoF). Sin embargo, con una interfaz común es posible realizar un reemplazo FIXME: drop-in del proxy en el código del cliente -el código del cliente está escrito para hablar al objeto original, y no necesita ser cambiado para aceptar el proxy (éste es probablemente el quiz principal de Proxy). Además, se fuerza a que Implementation complete, a través de la interfaz común, todas las funciones que Proxy necesita llamar.

La diferencia entre Proxy y Estado está en los problemas que pueden resolver. Los usos más comunes para Proxy que describe el GoF son:

1. Proxy remoto. Representan a objetos en un espacio de direcciones distinto. Lo implementan algunas tecnologías de objetos remotos.

2. Proxy virtual. Proporciona inicialización FIXME: vaga para crear objetos costosos bajo demanda.

3. Proxy de protección. Se usa cuando no se desea que el programador cliente tenga acceso completo al objecto representado.

4. Referencia inteligente. Para añadir acciones adicionales cuando se acceda al objeto representado. El conteo de referencias es un buen ejemplo: mantiene un registro del número de referencias que se mantienen para un objeto en particular, para implementar el FIXME: copy-on-write idiom y para prevenir el FIXME: object aliasing.

9.6.2. Estado: cambiar el comportamiento del objeto

El patrón Estado produce un objeto que parece que cambia su clase, y le será útil cuando descubra que tiene código condicional en todas o casi todas sus funciones. Al igual que Proxy, un Estado se crea teniendo un objeto front-end que usa un objeto back-end de implementación para completar sus tareas. Sin embargo, el patrón Estado alterna entre una implementación y otra durante la vida del objeto front-end, para mostrar un comportamiento distinto ante las mismas llamadas a función. Es una forma de mejorar la implementación de su código cuando realiza un montón de pruebas en cada una de sus funciones antes de decidir qué hacer con esa función. Por ejemplo, el cuento del príncipe convertido en rana contiene un objeto (la criatura) que se comporta de modo distinto dependiendo del estado en el que se encuentre. Podría implementar esto comprobando un `bool`:

```
//: C10:KissingPrincess.cpp
#include <iostream>
using namespace std;
```

```
class Creature {
  bool isFrog;
public:
  Creature() : isFrog(true) {}
  void greet() {
    if(isFrog)
      cout << "Ribbet!" << endl;
    else
      cout << "Darling!" << endl;
  }
  void kiss() { isFrog = false; }
};

int main() {
  Creature creature;
  creature.greet();
  creature.kiss();
  creature.greet();
} ///:~
```

Sin embargo, la función greet(), y cualquier otra función que tenga que comprobar isFrog antes de realizar sus operaciones, acaban con código poco elegante, especialmente cuando haya que añadir estados adicionales al sistema. Delegando las operaciones a un objeto Estado que puede cambiarse, el código se simplifica.

```
//: C10:KissingPrincess2.cpp
// The State pattern.
#include <iostream>
#include <string>
using namespace std;

class Creature {
  class State {
  public:
    virtual string response() = 0;
  };
  class Frog : public State {
  public:
    string response() { return "Ribbet!"; }
  };
  class Prince : public State {
  public:
    string response() { return "Darling!"; }
  };
  State* state;
public:
  Creature() : state(new Frog()) {}
  void greet() {
    cout << state->response() << endl;
  }
  void kiss() {
    delete state;
    state = new Prince();
  }
};
```

```
int main() {
  Creature creature;
  creature.greet();
  creature.kiss();
  creature.greet();
} ///:~
```

No es necesario hacer las clases FIXME: implementadoras anidadas ni privadas, pero si lo hace, el código será más limpio.

Note que los cambios en las clases Estado se propagan automáticamente por todo su código, en lugar de requerir una edición de las clases para efectuar los cambios.

9.7. Adaptador

Un Adaptador coge un tipo y genera una interfaz para algún otro tipo. Es útil cuando se tiene una librería o trozo de código que tiene una interfaz particular, y otra librería o trozo de código que usa las mismas ideas básicas que la primera librería, pero se expresa de forma diferente. Si se adaptan las formas de expresión entre sí, se puede crear una solución rápidamente.

Suponga que tiene una clase productora que genera los números de Fibonacci:

```
//: C10:FibonacciGenerator.h
#ifndef FIBONACCIGENERATOR_H
#define FIBONACCIGENERATOR_H

class FibonacciGenerator {
  int n;
  int val[2];
public:
  FibonacciGenerator() : n(0) { val[0] = val[1] = 0; }
  int operator()() {
    int result = n > 2 ? val[0] + val[1] : n > 0 ? 1 : 0;
    ++n;
    val[0] = val[1];
    val[1] = result;
    return result;
  }
  int count() { return n; }
};
#endif // FIBONACCIGENERATOR_H ///:~
```

Como es un productor, se usa llamando al operador(), de esta forma:

```
//: C10:FibonacciGeneratorTest.cpp
#include <iostream>
#include "FibonacciGenerator.h"
using namespace std;

int main() {
  FibonacciGenerator f;
  for(int i =0; i < 20; i++)
```

```
    cout << f.count() << ": " << f() << endl;
} ///:~
```

A lo mejor le gustaría coger este generador y realizar operaciones de algoritmos numéricos STL con él. Desafortunadamente, los algoritmos STL sólo trabajan con iteradores, así que tiene dos interfaces que no casan. La solución es crear un adaptador que coja el FibonacciGenerator y produzca un iterador para los algoritmos STL a usar. Dado que los algoritmos numéricos sólo necesitan un iterador de entrada, el Adaptador es bastante directo (para algo que produce un iterador STL, es decir):

```
//: C10:FibonacciAdapter.cpp
// Adapting an interface to something you already have.
#include <iostream>
#include <numeric>
#include "FibonacciGenerator.h"
#include "../C06/PrintSequence.h"
using namespace std;

class FibonacciAdapter { // Produce an iterator
  FibonacciGenerator f;
  int length;
public:
  FibonacciAdapter(int size) : length(size) {}
  class iterator;
  friend class iterator;
  class iterator : public std::iterator<
    std::input_iterator_tag, FibonacciAdapter, ptrdiff_t> {
    FibonacciAdapter& ap;
  public:
    typedef int value_type;
    iterator(FibonacciAdapter& a) : ap(a) {}
    bool operator==(const iterator&) const {
      return ap.f.count() == ap.length;
    }
    bool operator!=(const iterator& x) const {
      return !(*this == x);
    }
    int operator*() const { return ap.f(); }
    iterator& operator++() { return *this; }
    iterator operator++(int) { return *this; }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this); }
};

int main() {
  const int SZ = 20;
  FibonacciAdapter a1(SZ);
  cout << "accumulate: "
    << accumulate(a1.begin(), a1.end(), 0) << endl;
  FibonacciAdapter a2(SZ), a3(SZ);
  cout << "inner product: "
    << inner_product(a2.begin(), a2.end(), a3.begin(), 0)
    << endl;
  FibonacciAdapter a4(SZ);
  int r1[SZ] = {0};
```

```
  int* end = partial_sum(a4.begin(), a4.end(), r1);
  print(r1, end, "partial_sum", " ");
  FibonacciAdapter a5(SZ);
  int r2[SZ] = {0};
  end = adjacent_difference(a5.begin(), a5.end(), r2);
  print(r2, end, "adjacent_difference", " ");
} ///:~
```

Se inicializa un FibonacciAdapter diciéndole cuán largo puede ser la secuencia de Fibonacci. Cuando se crea un iterador, simplemente captura una referencia al FibonacciAdapter que lo contiene para que pueda acceder al FibonacciGenerator y la longitud. Observe que la comparación de equivalencia ignora el valor de la derecha, porque el único asunto importante es si el generador ha alcanzado su longitud. Además, el operator++() no modifica el iterador; la única operación que cambia el estado del FibonacciAdapter es llamar a la función operator() del generador en el FibonacciGenerator. Puede aceptarse esta versión extremadamente simple del iterador porque las restricciones de un Input Iterator son muy estrictas; concretamente, sólo se puede leer cada valor de la secuencia una vez.

En main(), puede verse que los cuatro tipos distintos de algoritmos numéricos se testan satisfactoriamente con el FibonacciAdapter.

9.8. Template Method

El marco de trabajo de una aplicación nos permite heredar de una clase o conjunto de ellas y crear una nueva aplicación, reutilizando la mayoría del código de las clases existentes y sobreescribiendo una o más funciones para adaptar la aplicación a nuestras necesidades.

Una característica importante de Template Method es que está definido en la clase base (a veces como una función privada) y no puede cambiarse -el Template Method es lo que permanece invariable. Llama a otras funciones de clase base (las que se sobreescriben) para hacer su trabajo, pero el programador cliente no es necesariamente capaz de llamarlo directamente, como puede verse aquí:

```
//: C10:TemplateMethod.cpp
// Simple demonstration of Template Method.
#include <iostream>
using namespace std;

class ApplicationFramework {
protected:
  virtual void customize1() = 0;
  virtual void customize2() = 0;
public:
  void templateMethod() {
    for(int i = 0; i < 5; i++) {
      customize1();
      customize2();
    }
  }
};

// Create a new "application":
```

```
class MyApp : public ApplicationFramework {
protected:
  void customize1() { cout << "Hello "; }
  void customize2() { cout << "World!" << endl; }
};

int main() {
  MyApp app;
  app.templateMethod();
} ///:~
```

El motor que ejecuta la aplicación es el Template Method. En una aplicación gráfica, este motor sería el bucle principal de eventos. El programador cliente simplemente proporciona las definiciones para customize1() y customize2(), y la aplicación está lista para ejecutarse.

9.9. Estrategia: elegir el algoritno en tiempo de ejecución

Observe que el Template Method es el código que no cambia, y las funciones que sobreescribe son el código cambiante. Sin embargo, este cambio está fijado en tiempo de compilación, a través de la herencia. Siguiendo la máxima de preferir composición a herencia, se puede usar una composición para aproximar el problema de separar código que cambia de código que permanece, y generar el patrón Estrategia. Esta aproximación tiene un beneficio único: en tiempo de ejecución se puede insertar el código que cambia. Estrategia también añade un Contexto que puede ser una clase sucedánea que controla la selección y uso del objeto estrategia -¡igual que Estado!.

Estrategia significa exactamente eso: se puede resolver un problema de muchas maneras. Imagine que ha olvidado el nombre de alguien. Estas son las diferentes maneras para lidiar con esa situación:

```
//: C10:Strategy.cpp
// The Strategy design pattern.
#include <iostream>
using namespace std;

class NameStrategy {
public:
  virtual void greet() = 0;
};

class SayHi : public NameStrategy {
public:
  void greet() {
    cout << "Hi! How's it going?" << endl;
  }
};

class Ignore : public NameStrategy {
public:
  void greet() {
    cout << "(Pretend I don't see you)" << endl;
```

```
  }
};

class Admission : public NameStrategy {
public:
  void greet() {
    cout << "I'm sorry. I forgot your name." << endl;
  }
};

// The "Context" controls the strategy:
class Context {
  NameStrategy& strategy;
public:
  Context(NameStrategy& strat) : strategy(strat) {}
  void greet() { strategy.greet(); }
};

int main() {
  SayHi sayhi;
  Ignore ignore;
  Admission admission;
  Context c1(sayhi), c2(ignore), c3(admission);
  c1.greet();
  c2.greet();
  c3.greet();
} ///:~
```

Normalmente, Context::greet() sería más complejo; es el análogo de Template Method porque contiene el código que no cambia. Pero puede ver en main() que la elección de la estrategia puede realizarse en tiempo de ejecución. Llendo un paso más allá, se puede combinar esto con el patrón Estado y cambiar la Estrategia durante el tiempo de vida del objeto Contexto.

9.10. Cadena de Responsabilidad: intentar una secuencia de estrategias

Debe pensarse en Cadena de Responsabilidad como en una generalización dinámica de la recursión, usando objetos Estrategia. Se hace una llamada y cada Estrategia de la secuencia intenta satisfacer la llamada. El proceso termina cuando una de las Estrategias tiene éxito o la cadena termina. En la recursión, una función se llama a sí misma una y otra vez hasta que se alcanza una condición de finalización; con Cadena de Responsabilidad, una función se llama a sí misma, la cual (moviendo la cadena de Estrategias) llama a una implementación diferente de la función, etc, hasta que se alcanza la condición de finalización. Dicha condición puede ser que se ha llegado al final de la cadena (lo que devuelve un objeto por defecto; puede que no sea capaz de proporcionar un resultado por defecto, así que debe ser capaz de determinar el éxito o fracaso de la cadena) o que una de las Estrategias ha tenido éxito.

En lugar de llamar a una única función para satisfacer una petición, hay múltiples funciones en la cadetna que tienen la oportunidad de hacerlo, de manera que tiene el aspecto de un sistema experto. Dado que la cadena es en la práctica una lista,

puede crearse dinámicamente, así que podría verse como una sentencia `switch` más general y construida dinámicamente.

En el GoF, hay bastante discusión sobre cómo crear la cadena de responsabilidad como una lista enlazada. Sin embargo, cuando se estudia el patrón, no debería importar cómo se crea la cadena; eso es un detalle de implementación. Como el GoF se escribió antes de que los contenedores STL estuvieran disponibles en la mayoría de los compiladores de C++, las razones más probables son (1) que no había listas incluídas y por lo tanto tenían que crear una y (2) que las estructuras de datos suelen verse como una habilidad fundamental en las Escuelas (o Facultades), y a los autores del GoF no se les ocurrió la idea de que las estructuras de datos fueran herramientas estándar disponibles junto con el lenguaje de programación.. Los detalles del contenedor usado para implementar la Cadena de Responsabilidad como una cadena (una lista enlazada en el GoF) no añaden nada a la solución, y puede implementarse usando un contenedor STL, como se muestra abajo.

Aquí puede ver una Cadena de Responsabilidad que encuentra automáticamente una solución usando un mecanismo para recorrer automática y recursivamente cada Estrategia de la cadena:

```
//: C10:ChainOfReponsibility.cpp
// The approach of the five-year-old.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

enum Answer { NO, YES };

class GimmeStrategy {
public:
  virtual Answer canIHave() = 0;
  virtual ~GimmeStrategy() {}
};

class AskMom : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Mooom? Can I have this?" << endl;
    return NO;
  }
};

class AskDad : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Dad, I really need this!" << endl;
    return NO;
  }
};

class AskGrandpa : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Grandpa, is it my birthday yet?" << endl;
    return NO;
  }
};
```

```
class AskGrandma : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Grandma, I really love you!" << endl;
    return YES;
  }
};

class Gimme : public GimmeStrategy {
  vector<GimmeStrategy*> chain;
public:
  Gimme() {
    chain.push_back(new AskMom());
    chain.push_back(new AskDad());
    chain.push_back(new AskGrandpa());
    chain.push_back(new AskGrandma());
  }
  Answer canIHave() {
    vector<GimmeStrategy*>::iterator it = chain.begin();
    while(it != chain.end())
      if((*it++)->canIHave() == YES)
        return YES;
    // Reached end without success...
    cout << "Whiiiiinnne!" << endl;
    return NO;
  }
  ~Gimme() { purge(chain); }
};

int main() {
  Gimme chain;
  chain.canIHave();
} ///:~
```

Observe que la clase de Contexto Gimme y todas las clases Estrategia derivan de la misma clase base, GimmeStrategy.

Si estudia la sección sobre Cadena de Responsabilidad del GoF, verá que la estructura difiere significativamente de la que se muestra más arriba, porque ellos se centran en crear su propia lista enlazada. Sin embargo, si mantiene en mente que la esencia de Cadena de Responsabilidad es probar muchas soluciones hasta que encuentre la que funciona, se dará cuenta de que la implementación del mecanismo de secuenciación no es parte esencial del patrón.

9.11. Factorías: encapsular la creación de objetos

Cuando se descubre que se necesitan añadir nuevos tipos a un sistema, el primer paso más sensato es usar polimorfismo para crear una interfaz común para esos nuevos tipos. Así, se separa el resto del código en el sistema del conocimiento de los tipos específicos que se están añadiendo. Los tipos nuevos pueden añadirse sin "molestar" al código existente, o eso parece. A primera vista, podría parecer que hace falta cambiar el código únicamente en los lugares donde se hereda un tipo nuevo, pero esto no es del todo cierto. Todavía hay que crear un objeto de este nuevo tipo, y en el momento de la creación hay que especificar qué constructor usar. Por lo tanto, si

el codigo que crea objetos está distribuido por toda la aplicación, se obtiene el mismo problema que cuando se añaden tipos -hay que localizar todos los puntos del código donde el tipo tiene importancia. Lo que imoporta es la creación del tipo, más que el uso del mismo (de eso se encarga el polimorfismo), pero el efecto es el mismo: añadir un nuevo tipo puede causar problemas.

La solución es forzar a que la creación de objetos se lleve a cabo a través de una factoría común, en lugar de permitir que el código creacional se disperse por el sistema. Si todo el código del programa debe ir a esta factoría cada vez que necesita crear uno de esos objetos, todo lo que hay que hacer para añadir un objeto es modificar la factoría. Este diseño es una variación del patrón conocido comúnmente como Factory Method. Dado que todo programa orientado a objetos crea objetos, y como es probable que haya que extender el programa añadiendo nuevos tipos, las factorías pueden ser el más útil de todos los patrones de diseño.

Como ejemplo, considere el ampliamente usado ejemplo de figura (Shape). Una aproximación para implementar una factoría es definir una función miembro estática en la clase base:

```
//: C10:ShapeFactory1.cpp
#include <iostream>
#include <stdexcept>
#include <cstddef>
#include <string>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  static Shape* factory(const string& type)
    throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle() {} // Private constructor
  friend class Shape;
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square() {}
  friend class Shape;
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  ~Square() { cout << "Square::~Square" << endl; }
```

```
};

Shape* Shape::factory(const string& type)
  throw(Shape::BadShapeCreation) {
  if(type == "Circle") return new Circle;
  if(type == "Square") return new Square;
  throw BadShapeCreation(type);
}

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(Shape::factory(sl[i]));
  } catch(Shape::BadShapeCreation e) {
    cout << e.what() << endl;
    purge(shapes);
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    shapes[i]->erase();
  }
  purge(shapes);
} ///:~
```

La función factory() toma un argumento que le permite determinar qué tipo de figura crear. Aquí, el argumento es una cadena, pero podría ser cualquier conjunto de datos. El método factory() es el único código del sistema que hay que cambiar cuando se añade un nuevo tipo de figura. (Los datos de inicialización para los objetos vendrán supuestamente de algún sitio fuera del sistema y no serán un FIXME: hard-coded array como en el ejemplo.)

Para asegurar que la creación sólo puede realizarse en factory(), los constructores de cada tipo específico de figura se hacen privados, y Shape se declara como `friend` de forma que factory() tiene acceso a los mismos. (También se podría declarar sólamente Shape::factory() como `friend`, pero parece razonablemente inocuo declarar la clase base entera.) Hay otra implicación importante de este diseño -la clase base, Shape, debe conocer ahora los detalles de todas las clases derivadas -una propiedad que el diseño orientado a objetos intenta evitar. Para `frameworks` o cualquier librería de clases que deban poder extenderse, esto hace que se convierta rápidamente en algo difícil de manejar, ya que la clase base debe actualizarse en cuanto se añada un tipo nuevo a la jerarquía. Las factorías polimórficas, descritas en la siguiente subsección, se pueden usar para evitar esta dependencia circular tan poco deseada.

9.11.1. Factorías polimórficas

La función estática factory() en el ejemplo anterior fuerza que las operaciones de creación se centren en un punto, de forma que sea el único sitio en el que haya que cambiar código. Esto es, sin duda, una solución razonable, ya que encapsula amablemente el proceso de crear objetos. Sin embargo, el GoF enfatiza que la razón de

ser del patrón Factory Method es que diferentes tipos de factorías se puedan derivar de la factoría básica. Factory Method es, de hecho, un tipo especial de factoría polimórfica. Esto es ShapeFactory1.cpp modificado para que los Factory Methods estén en una clase aparte como funciones virtuales.

```
//: C10:ShapeFactory2.cpp
// Polymorphic Factory Methods.
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <stdexcept>
#include <cstddef>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class ShapeFactory {
  virtual Shape* create() = 0;
  static map<string, ShapeFactory*> factories;
public:
  virtual ~ShapeFactory() {}
  friend class ShapeFactoryInitializer;
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  static Shape*
  createShape(const string& id) throw(BadShapeCreation) {
    if(factories.find(id) != factories.end())
      return factories[id]->create();
    else
      throw BadShapeCreation(id);
  }
};

// Define the static object:
map<string, ShapeFactory*> ShapeFactory::factories;

class Circle : public Shape {
  Circle() {} // Private constructor
  friend class ShapeFactoryInitializer;
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Circle; }
    friend class ShapeFactoryInitializer;
  };
public:
  void draw() { cout << "Circle::draw" << endl; }
```

```
  void erase() { cout << "Circle::erase" << endl; }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square() {}
  friend class ShapeFactoryInitializer;
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Square; }
    friend class ShapeFactoryInitializer;
  };
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  ~Square() { cout << "Square::~Square" << endl; }
};

// Singleton to initialize the ShapeFactory:
class ShapeFactoryInitializer {
  static ShapeFactoryInitializer si;
  ShapeFactoryInitializer() {
    ShapeFactory::factories["Circle"]= new Circle::Factory;
    ShapeFactory::factories["Square"]= new Square::Factory;
  }
  ~ShapeFactoryInitializer() {
    map<string, ShapeFactory*>::iterator it =
      ShapeFactory::factories.begin();
    while(it != ShapeFactory::factories.end())
      delete it++->second;
  }
};

// Static member definition:
ShapeFactoryInitializer ShapeFactoryInitializer::si;

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(ShapeFactory::createShape(sl[i]));
  } catch(ShapeFactory::BadShapeCreation e) {
    cout << e.what() << endl;
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    shapes[i]->erase();
  }
  purge(shapes);
} ///:~
```

Ahora, Factory Method aparece en su propia clase, ShapeFactory, como `virtual create()`. Es una función miembro privada, lo que significa que no puede ser llamada directametne, pero puede ser sobreescrita. Las subclases de Shape deben crear cada una su propias subclases de ShapeFactory y sobreescribir el método create para crear un objeto de su propio tipe. Estas factorías son privadas, de forma que sólo pueden ser accedidas desde el Factory Method principal. De esta forma, todo el código cliente debe pasar a través del Factory Method para crear objetos.

La verdadera creación de figuras se realiza llamando a ShapeFactory::createShape( ), que es una función estática que usa el mapa en ShapeFactory para encontrar la objeto factoría apropiado basándose en el identificador que se le pasa. La factoría crea el objeto figura directamente, pero podría imaginarse un problema más complejo en el que el objeto factoría apropiado se devuelve y luego lo usa quien lo ha llamado para crear un objeto de una manera más sofisticada. Sin embargo, parece que la mayoría del tiempo no hacen falta las complejidades del Factory Method polimórfico, y bastará con una única función estática en la clase base (como se muestra en ShapeFactory1.cpp).

Observe que el ShapeFactory debe ser inicializado cargando su mapa con objetos factory, lo que tiene lugar en el Singleton ShapeFactoryInitializer. Así que para añadir un nuevo tipo a este diseño debe definir el tipo, crear una factoría, y modificar ShapeFactoryInitializer para que se inserte una instancia de su factoría en el mapa. Esta complejidad extra, sugiere de nuevo el uso de un Factory Method estático si no necesita crear objetos factoría individuales.

9.11.2. Factorías abstractas

El patrón Factoría Abstracta se parece a las factorías que hemos visto anteriormente, pero con varios Factory Methods. Cada uno de los Factory Method crea una clase distinta de objeto. Cuando se crea el objecto factoría, se decide cómo se usarán todos los objetos creados con esa factoría. El ejemplo del GoF implementa la portabilidad a través de varias interfaces gráficas de usuario (GUI): se crea el objeto factoría apropiado para la GUI con la que se está trabajando y desde ahí en adelante, cuando le pida un menú, botón, barra deslizante y demás, creará automáticamente la versión apropiada para la GUI de ese elemento. Por lo tanto, es posible aislar, en un solo lugar, el efecto de cambiar de una GUI a otra.

Por ejemplo, suponga que está creando un entorno para juegos de propósito general y quiere ser capaz de soportar diferentes tipos de juegos. Así es como sería usando una Factoría Abstracta:

```
//: C10:AbstractFactory.cpp
// A gaming environment.
#include <iostream>
using namespace std;

class Obstacle {
public:
  virtual void action() = 0;
};

class Player {
public:
  virtual void interactWith(Obstacle*) = 0;
};
```

```
class Kitty: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "Kitty has encountered a ";
    ob->action();
  }
};

class KungFuGuy: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "KungFuGuy now battles against a ";
    ob->action();
  }
};

class Puzzle: public Obstacle {
public:
  void action() { cout << "Puzzle" << endl; }
};

class NastyWeapon: public Obstacle {
public:
  void action() { cout << "NastyWeapon" << endl; }
};

// The abstract factory:
class GameElementFactory {
public:
  virtual Player* makePlayer() = 0;
  virtual Obstacle* makeObstacle() = 0;
};

// Concrete factories:
class KittiesAndPuzzles : public GameElementFactory {
public:
  virtual Player* makePlayer() { return new Kitty; }
  virtual Obstacle* makeObstacle() { return new Puzzle; }
};

class KillAndDismember : public GameElementFactory {
public:
  virtual Player* makePlayer() { return new KungFuGuy; }
  virtual Obstacle* makeObstacle() {
    return new NastyWeapon;
  }
};

class GameEnvironment {
  GameElementFactory* gef;
  Player* p;
  Obstacle* ob;
public:
  GameEnvironment(GameElementFactory* factory)
  : gef(factory), p(factory->makePlayer()),
    ob(factory->makeObstacle()) {}
  void play() { p->interactWith(ob); }
  ~GameEnvironment() {
    delete p;
    delete ob;
```

```
    delete gef;
  }
};

int main() {
  GameEnvironment
    g1(new KittiesAndPuzzles),
    g2(new KillAndDismember);
  g1.play();
  g2.play();
}
/* Output:
Kitty has encountered a Puzzle
KungFuGuy now battles against a NastyWeapon */ ///:~
```

En este entorno, los objetos Player interactúan con objetos Obstacle, pero los tipos de los jugadores y los obstáculos dependen del juego. El tipo de juego se determina eligiendo un GameElementFactory concreto, y luego el GameEnvironment controla la configuración y ejecución del juego. En este ejemplo, la configuración y ejecución son simples, pero dichas actividades (las condiciones iniciales y los cambios de estado) pueden determinar gran parte del resultado del juego. Aquí, GameEnvironment no está diseñado para ser heredado, aunque puede tener sentido hacerlo.

Este ejemplo también ilustra el despachado doble, que se explicará más adelante.

9.11.3. Constructores virtuales

```
//: C10:VirtualConstructor.cpp
#include <iostream>
#include <string>
#include <stdexcept>
#include <stdexcept>
#include <cstddef>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
  Shape* s;
  // Prevent copy-construction & operator=
  Shape(Shape&);
  Shape operator=(Shape&);
protected:
  Shape() { s = 0; }
public:
  virtual void draw() { s->draw(); }
  virtual void erase() { s->erase(); }
  virtual void test() { s->test(); }
  virtual ~Shape() {
    cout << "~Shape" << endl;
    if(s) {
      cout << "Making virtual call: ";
      s->erase(); // Virtual call
    }
    cout << "delete s: ";
```

```
    delete s; // The polymorphic deletion
    // (delete 0 is legal; it produces a no-op)
  }
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  Shape(string type) throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle(Circle&);
  Circle operator=(Circle&);
  Circle() {} // Private constructor
  friend class Shape;
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  void test() { draw(); }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square(Square&);
  Square operator=(Square&);
  Square() {}
  friend class Shape;
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  void test() { draw(); }
  ~Square() { cout << "Square::~Square" << endl; }
};

Shape::Shape(string type) throw(Shape::BadShapeCreation) {
  if(type == "Circle")
    s = new Circle;
  else if(type == "Square")
    s = new Square;
  else throw BadShapeCreation(type);
  draw();  // Virtual call in the constructor
}

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  cout << "virtual constructor calls:" << endl;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(new Shape(sl[i]));
  } catch(Shape::BadShapeCreation e) {
    cout << e.what() << endl;
    purge(shapes);
    return EXIT_FAILURE;
  }
```

```
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    cout << "test" << endl;
    shapes[i]->test();
    cout << "end test" << endl;
    shapes[i]->erase();
  }
  Shape c("Circle"); // Create on the stack
  cout << "destructor calls:" << endl;
  purge(shapes);
} ///:~
```

9.12. Builder: creación de objetos complejos

```
//: C10:Bicycle.h
// Defines classes to build bicycles;
// Illustrates the Builder design pattern.
#ifndef BICYCLE_H
#define BICYCLE_H
#include <iostream>
#include <string>
#include <vector>
#include <cstddef>
#include "../purge.h"
using std::size_t;

class BicyclePart {
public:
  enum BPart { FRAME, WHEEL, SEAT, DERAILLEUR,
    HANDLEBAR, SPROCKET, RACK, SHOCK, NPARTS };
private:
  BPart id;
  static std::string names[NPARTS];
public:
  BicyclePart(BPart bp) { id = bp; }
  friend std::ostream&
  operator<<(std::ostream& os, const BicyclePart& bp) {
    return os << bp.names[bp.id];
  }
};

class Bicycle {
  std::vector<BicyclePart*> parts;
public:
  ~Bicycle() { purge(parts); }
  void addPart(BicyclePart* bp) { parts.push_back(bp); }
  friend std::ostream&
  operator<<(std::ostream& os, const Bicycle& b) {
    os << "{ ";
    for(size_t i = 0; i < b.parts.size(); ++i)
      os << *b.parts[i] << ' ';
    return os << '}';
  }
};
```

```
class BicycleBuilder {
protected:
  Bicycle* product;
public:
  BicycleBuilder() { product = 0; }
  void createProduct() { product = new Bicycle; }
  virtual void buildFrame() = 0;
  virtual void buildWheel() = 0;
  virtual void buildSeat() = 0;
  virtual void buildDerailleur() = 0;
  virtual void buildHandlebar() = 0;
  virtual void buildSprocket() = 0;
  virtual void buildRack() = 0;
  virtual void buildShock() = 0;
  virtual std::string getBikeName() const = 0;
  Bicycle* getProduct() {
    Bicycle* temp = product;
    product = 0;  // Relinquish product
    return temp;
  }
};

class MountainBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "MountainBike";}
};

class TouringBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "TouringBike"; }
};

class RacingBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
```

```
  void buildShock();
  std::string getBikeName() const { return "RacingBike"; }
};

class BicycleTechnician {
  BicycleBuilder* builder;
public:
  BicycleTechnician() { builder = 0; }
  void setBuilder(BicycleBuilder* b) { builder = b; }
  void construct();
};
#endif // BICYCLE_H ///:~
```

```
//: C10:Bicycle.cpp {O} {-mwcc}
#include "Bicycle.h"
#include <cassert>
#include <cstddef>
using namespace std;

std::string BicyclePart::names[NPARTS] = {
  "Frame", "Wheel", "Seat", "Derailleur",
  "Handlebar", "Sprocket", "Rack", "Shock" };

// MountainBikeBuilder implementation
void MountainBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void MountainBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void MountainBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void MountainBikeBuilder::buildDerailleur() {
  product->addPart(
    new BicyclePart(BicyclePart::DERAILLEUR));
}
void MountainBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void MountainBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void MountainBikeBuilder::buildRack() {}
void MountainBikeBuilder::buildShock() {
  product->addPart(new BicyclePart(BicyclePart::SHOCK));
}

// TouringBikeBuilder implementation
void TouringBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void TouringBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
```

```
void TouringBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void TouringBikeBuilder::buildDerailleur() {
  product->addPart(
    new BicyclePart(BicyclePart::DERAILLEUR));
}
void TouringBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void TouringBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void TouringBikeBuilder::buildRack() {
  product->addPart(new BicyclePart(BicyclePart::RACK));
}
void TouringBikeBuilder::buildShock() {}

// RacingBikeBuilder implementation
void RacingBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void RacingBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void RacingBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void RacingBikeBuilder::buildDerailleur() {}
void RacingBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void RacingBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void RacingBikeBuilder::buildRack() {}
void RacingBikeBuilder::buildShock() {}

// BicycleTechnician implementation
void BicycleTechnician::construct() {
  assert(builder);
  builder->createProduct();
  builder->buildFrame();
  builder->buildWheel();
  builder->buildSeat();
  builder->buildDerailleur();
  builder->buildHandlebar();
  builder->buildSprocket();
  builder->buildRack();
  builder->buildShock();
} ///:~
```

```
//: C10:BuildBicycles.cpp
//{L} Bicycle
```

```
// The Builder design pattern.
#include <cstddef>
#include <iostream>
#include <map>
#include <vector>
#include "Bicycle.h"
#include "../purge.h"
using namespace std;

// Constructs a bike via a concrete builder
Bicycle* buildMeABike(
  BicycleTechnician& t, BicycleBuilder* builder) {
  t.setBuilder(builder);
  t.construct();
  Bicycle* b = builder->getProduct();
  cout << "Built a " << builder->getBikeName() << endl;
  return b;
}

int main() {
  // Create an order for some bicycles
  map <string, size_t> order;
  order["mountain"] = 2;
  order["touring"] = 1;
  order["racing"] = 3;

  // Build bikes
  vector<Bicycle*> bikes;
  BicycleBuilder* m = new MountainBikeBuilder;
  BicycleBuilder* t = new TouringBikeBuilder;
  BicycleBuilder* r = new RacingBikeBuilder;
  BicycleTechnician tech;
  map<string, size_t>::iterator it = order.begin();
  while(it != order.end()) {
    BicycleBuilder* builder;
    if(it->first == "mountain")
      builder = m;
    else if(it->first == "touring")
      builder = t;
    else if(it->first == "racing")
      builder = r;
    for(size_t i = 0; i < it->second; ++i)
      bikes.push_back(buildMeABike(tech, builder));
    ++it;
  }
  delete m;
  delete t;
  delete r;

  // Display inventory
  for(size_t i = 0; i < bikes.size(); ++i)
    cout << "Bicycle: " << *bikes[i] << endl;
  purge(bikes);
}

/* Output:
Built a MountainBike
Built a MountainBike
```

```
Built a RacingBike
Built a RacingBike
Built a RacingBike
Built a TouringBike
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Rack }
*/ ///:~
```

9.13. Observador

```
//: C10:Observer.h
// The Observer interface.
#ifndef OBSERVER_H
#define OBSERVER_H

class Observable;
class Argument {};

class Observer {
public:
  // Called by the observed object, whenever
  // the observed object is changed:
  virtual void update(Observable* o, Argument* arg) = 0;
  virtual ~Observer() {}
};
#endif // OBSERVER_H ///:~
```

```
//: C10:Observable.h
// The Observable class.
#ifndef OBSERVABLE_H
#define OBSERVABLE_H
#include <set>
#include "Observer.h"

class Observable {
  bool changed;
  std::set<Observer*> observers;
protected:
  virtual void setChanged() { changed = true; }
  virtual void clearChanged() { changed = false; }
public:
  virtual void addObserver(Observer& o) {
    observers.insert(&o);
  }
  virtual void deleteObserver(Observer& o) {
```

```
    observers.erase(&o);
  }
  virtual void deleteObservers() {
    observers.clear();
  }
  virtual int countObservers() {
    return observers.size();
  }
  virtual bool hasChanged() { return changed; }
  // If this object has changed, notify all
  // of its observers:
  virtual void notifyObservers(Argument* arg = 0) {
    if(!hasChanged()) return;
    clearChanged(); // Not "changed" anymore
    std::set<Observer*>::iterator it;
    for(it = observers.begin();it != observers.end(); it++)
      (*it)->update(this, arg);
  }
  virtual ~Observable() {}
};
#endif // OBSERVABLE_H ///:~
```

```
//: C10:InnerClassIdiom.cpp
// Example of the "inner class" idiom.
#include <iostream>
#include <string>
using namespace std;

class Poingable {
public:
  virtual void poing() = 0;
};

void callPoing(Poingable& p) {
  p.poing();
}

class Bingable {
public:
  virtual void bing() = 0;
};

void callBing(Bingable& b) {
  b.bing();
}

class Outer {
  string name;
  // Define one inner class:
  class Inner1;
  friend class Outer::Inner1;
  class Inner1 : public Poingable {
    Outer* parent;
  public:
    Inner1(Outer* p) : parent(p) {}
    void poing() {
```

```
      cout << "poing called for "
        << parent->name << endl;
      // Accesses data in the outer class object
    }
  } inner1;
  // Define a second inner class:
  class Inner2;
  friend class Outer::Inner2;
  class Inner2 : public Bingable {
    Outer* parent;
  public:
    Inner2(Outer* p) : parent(p) {}
    void bing() {
      cout << "bing called for "
        << parent->name << endl;
    }
  } inner2;
public:
  Outer(const string& nm)
  : name(nm), inner1(this), inner2(this) {}
  // Return reference to interfaces
  // implemented by the inner classes:
  operator Poingable&() { return inner1; }
  operator Bingable&() { return inner2; }
};

int main() {
  Outer x("Ping Pong");
  // Like upcasting to multiple base types!:
  callPoing(x);
  callBing(x);
} ///:~
```

9.13.1. El ejemplo de observador

```
//: C10:ObservedFlower.cpp
// Demonstration of "observer" pattern.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include "Observable.h"
using namespace std;

class Flower {
  bool isOpen;
public:
  Flower() : isOpen(false),
    openNotifier(this), closeNotifier(this) {}
  void open() { // Opens its petals
    isOpen = true;
    openNotifier.notifyObservers();
    closeNotifier.open();
  }
  void close() { // Closes its petals
```

```
    isOpen = false;
    closeNotifier.notifyObservers();
    openNotifier.close();
  }
  // Using the "inner class" idiom:
  class OpenNotifier;
  friend class Flower::OpenNotifier;
  class OpenNotifier : public Observable {
    Flower* parent;
    bool alreadyOpen;
  public:
    OpenNotifier(Flower* f) : parent(f),
      alreadyOpen(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(parent->isOpen && !alreadyOpen) {
        setChanged();
        Observable::notifyObservers();
        alreadyOpen = true;
      }
    }
    void close() { alreadyOpen = false; }
  } openNotifier;
  class CloseNotifier;
  friend class Flower::CloseNotifier;
  class CloseNotifier : public Observable {
    Flower* parent;
    bool alreadyClosed;
  public:
    CloseNotifier(Flower* f) : parent(f),
      alreadyClosed(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(!parent->isOpen && !alreadyClosed) {
        setChanged();
        Observable::notifyObservers();
        alreadyClosed = true;
      }
    }
    void open() { alreadyClosed = false; }
  } closeNotifier;
};

class Bee {
  string name;
  // An "inner class" for observing openings:
  class OpenObserver;
  friend class Bee::OpenObserver;
  class OpenObserver : public Observer {
    Bee* parent;
  public:
    OpenObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s breakfast time!" << endl;
    }
  } openObsrv;
  // Another "inner class" for closings:
  class CloseObserver;
  friend class Bee::CloseObserver;
```

```
  class CloseObserver : public Observer {
    Bee* parent;
  public:
    CloseObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s bed time!" << endl;
    }
  } closeObsrv;
public:
  Bee(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

class Hummingbird {
  string name;
  class OpenObserver;
  friend class Hummingbird::OpenObserver;
  class OpenObserver : public Observer {
    Hummingbird* parent;
  public:
    OpenObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s breakfast time!" << endl;
    }
  } openObsrv;
  class CloseObserver;
  friend class Hummingbird::CloseObserver;
  class CloseObserver : public Observer {
    Hummingbird* parent;
  public:
    CloseObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s bed time!" << endl;
    }
  } closeObsrv;
public:
  Hummingbird(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

int main() {
  Flower f;
  Bee ba("A"), bb("B");
  Hummingbird ha("A"), hb("B");
  f.openNotifier.addObserver(ha.openObserver());
  f.openNotifier.addObserver(hb.openObserver());
  f.openNotifier.addObserver(ba.openObserver());
  f.openNotifier.addObserver(bb.openObserver());
  f.closeNotifier.addObserver(ha.closeObserver());
  f.closeNotifier.addObserver(hb.closeObserver());
  f.closeNotifier.addObserver(ba.closeObserver());
```

```
  f.closeNotifier.addObserver(bb.closeObserver());
  // Hummingbird B decides to sleep in:
  f.openNotifier.deleteObserver(hb.openObserver());
  // Something changes that interests observers:
  f.open();
  f.open(); // It's already open, no change.
  // Bee A doesn't want to go to bed:
  f.closeNotifier.deleteObserver(
    ba.closeObserver());
  f.close();
  f.close(); // It's already closed; no change
  f.openNotifier.deleteObservers();
  f.open();
  f.close();
} ///:~
```

9.14. Despachado múltiple

```
//: C10:PaperScissorsRock.cpp
// Demonstration of multiple dispatching.
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Paper;
class Scissors;
class Rock;

enum Outcome { WIN, LOSE, DRAW };

ostream& operator<<(ostream& os, const Outcome out) {
  switch(out) {
    default:
    case WIN: return os << "win";
    case LOSE: return os << "lose";
    case DRAW: return os << "draw";
  }
}

class Item {
public:
  virtual Outcome compete(const Item*) = 0;
  virtual Outcome eval(const Paper*) const = 0;
  virtual Outcome eval(const Scissors*) const= 0;
  virtual Outcome eval(const Rock*) const = 0;
  virtual ostream& print(ostream& os) const = 0;
  virtual ~Item() {}
  friend ostream& operator<<(ostream& os, const Item* it) {
    return it->print(os);
```

```
  }
};

class Paper : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return DRAW; }
  Outcome eval(const Scissors*) const { return WIN; }
  Outcome eval(const Rock*) const { return LOSE; }
  ostream& print(ostream& os) const {
    return os << "Paper   ";
  }
};

class Scissors : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return LOSE; }
  Outcome eval(const Scissors*) const { return DRAW; }
  Outcome eval(const Rock*) const { return WIN; }
  ostream& print(ostream& os) const {
    return os << "Scissors";
  }
};

class Rock : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return WIN; }
  Outcome eval(const Scissors*) const { return LOSE; }
  Outcome eval(const Rock*) const { return DRAW; }
  ostream& print(ostream& os) const {
    return os << "Rock    ";
  }
};

struct ItemGen {
  Item* operator()() {
    switch(rand() % 3) {
      default:
      case 0: return new Scissors;
      case 1: return new Paper;
      case 2: return new Rock;
    }
  }
};

struct Compete {
  Outcome operator()(Item* a, Item* b) {
    cout << a << "\t" << b << "\t";
    return a->compete(b);
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  const int sz = 20;
  vector<Item*> v(sz*2);
```

```
  generate(v.begin(), v.end(), ItemGen());
  transform(v.begin(), v.begin() + sz,
    v.begin() + sz,
    ostream_iterator<Outcome>(cout, "\n"),
    Compete());
  purge(v);
} ///:~
```

9.14.1. Despachado múltiple con Visitor

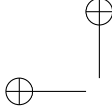
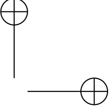
```
//: C10:BeeAndFlowers.cpp
// Demonstration of "visitor" pattern.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Gladiolus;
class Renuculus;
class Chrysanthemum;

class Visitor {
public:
  virtual void visit(Gladiolus* f) = 0;
  virtual void visit(Renuculus* f) = 0;
  virtual void visit(Chrysanthemum* f) = 0;
  virtual ~Visitor() {}
};

class Flower {
public:
  virtual void accept(Visitor&) = 0;
  virtual ~Flower() {}
};

class Gladiolus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Renuculus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Chrysanthemum : public Flower {
public:
```

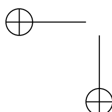
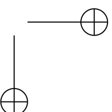
```
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

// Add the ability to produce a string:
class StringVal : public Visitor {
  string s;
public:
  operator const string&() { return s; }
  virtual void visit(Gladiolus*) {
    s = "Gladiolus";
  }
  virtual void visit(Renuculus*) {
    s = "Renuculus";
  }
  virtual void visit(Chrysanthemum*) {
    s = "Chrysanthemum";
  }
};

// Add the ability to do "Bee" activities:
class Bee : public Visitor {
public:
  virtual void visit(Gladiolus*) {
    cout << "Bee and Gladiolus" << endl;
  }
  virtual void visit(Renuculus*) {
    cout << "Bee and Renuculus" << endl;
  }
  virtual void visit(Chrysanthemum*) {
    cout << "Bee and Chrysanthemum" << endl;
  }
};

struct FlowerGen {
  Flower* operator()() {
    switch(rand() % 3) {
      default:
      case 0: return new Gladiolus;
      case 1: return new Renuculus;
      case 2: return new Chrysanthemum;
    }
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  vector<Flower*> v(10);
  generate(v.begin(), v.end(), FlowerGen());
  vector<Flower*>::iterator it;
  // It's almost as if I added a virtual function
  // to produce a Flower string representation:
  StringVal sval;
  for(it = v.begin(); it != v.end(); it++) {
    (*it)->accept(sval);
    cout << string(sval) << endl;
  }
```

```
    // Perform "Bee" operation on all Flowers:
    Bee bee;
    for(it = v.begin(); it != v.end(); it++)
      (*it)->accept(bee);
    purge(v);
  } ///:~
```

9.15. Resumen

9.16. Ejercicios

10: Concurrencia

Los objetos ofrecen una forma de dividir un programa en diferentes secciones. A menudo, también es necesario dividir un programa, independientemente de las subtareas en ejecución.

Utilizando multihilado, un hilo de ejecución dirige cada una esas subtareas independientes, y puedes programar como si cada hilo tuviera su propia CPU. Un mecanismo interno reparte el tiempo de CPU por ti, pero en general, no necesitas pensar acerca de eso, lo que ayuda a simplificar la programación con múltiples hilos.

Un proceso es un programa autocontenido en ejecución con su propio espacio de direcciones. Un sistema operativo multitarea puede ejecutar más de un proceso (programa) en el mismo tiempo, mientras ....., por medio de cambios periódicos de CPU de una tarea a otra. Un hilo es una simple flujo de control secuencial con un proceso. Un proceso puede tener de este modo múltiples hilos en ejecución concurrentes. Puesto que los hilos se ejecutan con un proceso simple, pueden compartir memoria y otros recursos. La dificultad fundamental de escribir programas multihilados está en coordinar el uso de esos recursos entre los diferentes hilos.

Hay muchas aplicaciones posibles para el multihilado, pero lo más usual es querer usarlo cuando tienes alguna parte de tu programa vinculada a un evento o recurso particular. Para evitar bloquear el resto de tu programa, creas un hilo asociado a ese evento o recurso y le permites ejecutarse independientemente del programa principal.

La programación concurrente se como caminar en un mundo completamente nuevo y aprender un nuevo lenguaje de programación, o por lo menos un nuevo conjunto de conceptos del lenguaje. Con la aparición del soporte para los hilos en la mayoría de los sistemas operativos para microcomputadores, han aparecido también en los lenguajes de programación o librerías extensiones para los hilos. En cualquier caso, programación hilada:

1. Parece misteriosa y requiere un esfuerzo en la forma de pensar acerca de la programación.

2. En otros lenguajes el soporte a los hilos es similar. Cuando entiendas los hilos, comprenderás una jerga común.

Comprender la programación concurrente está al mismo nivel de dificultad que comprender el polimorfismo. Si pones un poco de esfuerzo, podrás entender el mecanismo básico, pero generalmente necesitará de un entendimiento y estudio profundo para desarrollar una comprensión auténtica sobre el tema. La meta de este capítulo es darte una base sólida en los principios de concurrencia para que puedas entender los conceptos y escribir programas multihilados razonables. Sé consciente de que puedes confiarte fácilmente. Si vas a escribir algo complejo, necesitarás estudiar libros específicos sobre el tema.

10.1. Motivación

Una de las razones más convincentes para usar concurrencia es crear una interfaz sensible al usuario. Considera un programa que realiza una operación de CPU intensiva y, de esta forma, termina ignorando la entrada del usuario y comienza a no responder. El programa necesita continuar controlandor sus operaciones, y al mismo tiempo necesita devolver el control al botón de la interfaz de usuario para que el programa pueda responder al usuario. Si tienes un botón de "Salir", no querrás estar forzado a sondearlo en todas las partes de código que escribas en tu programa. (Esto acoplaría tu botón de salir a lo largo del programa y sería un quebradero de cabeza a la hora de mantenerlo). .....

Una función convencional no puede continuar realizando sus operaciones y al mismo tiempo devolver el control al resto del programa. De hecho, suena a imposible, como si la CPU estuviera en dos lugares a la vez, pero esto es precisamente la "ilusión" que la concurrencia permite (en el caso de un sistema multiprocesador, debe haber más de una "ilusión").

También puedes usar concurrencia para optimizar la carga de trabajo. Por ejemplo, podrías necesitar hacer algo importante mientras estás estancado esperando la llegada de una entrada del puerto I/O. Sin hilos, la única solución razonable es sondear los puertos I/O, que es costoso y puede ser difícil.

Si tienes una máquina multiprocesador, los múltiples hilos pueden ser distribuídos a lo largo de los múltiples procesadores, pudiendo mejorar considerablemente la carga de trabajo. Este es el típico caso de los potentes servidores web multiprocesdor, que pueden distribuir un gran número de peticiones de usuario por todas las CPUs en un programa que asigna un hilo por petición.

Un programa que usa hilos en una máquina monoprocesador hará una cosa en un tiempo dado, por lo que es teóricamente posible escribir el mismo programa sin el uso de hilos. Sin embargo, el multihilado proporciona un beneficio de optimización importante: El diseño de un programa puede ser maravillosamente simple. Algunos tipos de problemas, como la simulación - un video juego, por ejemplo - son difíciles de resolver sin el soporte de la concurrencia.

El modelo hilado es una comodidad de la programación para simplificar el manejo de muchas operaciones al mismo tiempo con un simple programa: La CPU desapilará y dará a cada hilo algo de su tiempo. Cada hilo tiene consciencia de que tiene un tiempo constante de uso de CPU, pero el tiempo de CPU está actualmente repartido entre todo los hilos. La excepción es un programa que se ejecuta sobre múltiples CPU's. Pero una de las cosas fabulosas que tiene el hilado es que te abstrae de esta capa, por lo que tu código no necesita saber si está ejecutándose sobre una sóla CPU o sobre varias.[149] De este modo, usar hilos es una manera de crear programas escalables de forma transparente - si un programa se está ejecutando demasiado despacio, puedes acelerarlo fácilmente añadiendo CPUs a tu ordenador. La multitarea y el multihilado tienden a ser las mejores opciones a utilizar en un sistema multiprocesador.

El uso de hilos puede reducir la eficiencia computacional un poco, pero el aumento neto en el diseño del programa, balanceo de recursos, y la comodidad del usuario a menudo es más valorado. En general, los hilos te permiten crear diseñor más desacoplados; de lo contrario, las partes de tu código estaría obligadas a prestar atención a tareas que podrías manejarlas con hilos normalmente.

10.2. Concurrencia en C++

Cuando el Comité de Estándares de C++ estaba creando el estándar inicial de C++, el mecanismo de concurrencia fue excluído de forma explícita porque C no tenía uno y también porque había varíos enfoques rivales acerca de su implementación. Parecía demasiado restrictivo forzar a los programadores a usar una sola alternativa.

Sin embargo, la alternativa resultó ser peor. Para usar concurriencia, tenías que encontrar y aprender una librería y ocuparte de su indiosincrasia y las incertidumbres de trabajar con un vendedor particular. Además, no había garantía de que una librería funcionaría en diferentes compiladores o en distintas plataformas. También, desde que la concurrencia no formaba parte del estándar del lenguaje, fue más difícil encontrar programadores C++ que también entendieran la programación concurrente.

Otra influencia pudo ser el lenguaje Java, que incluyó concurrencia en el núcleo del lenguaje. Aunque el multihilado is aún complicado, los programadores de Java tienden a empezar a aprenderlo y usarlo desde el principio.

El Comité de Estándares de C++ está considerando incluir el soporte a la concurrencia en la siguiente iteración de C++, pero en el momento de este escrito no estaba claro qué aspecto tendrá la librería. Decidimos usar la librería ZThread como base para este capítulo. La escogimos por su diseño, y es open-source y gratuitamente descargable desde http://zthread.sourceforge.net. Eric Crahen de IBM, el autor de la librería ZThread, fue decisivo para crear este capítulo.[150]

Este capítulo utiliza sólo un subgrupo de la librería ZThread, de acuerdo con el convenio de ideas fundamentales sobre los hilos. La librería ZThread contiene un soporte a los hilos significativamente más sofisticado que el que se muestra aquí, y deberías estudiar esa librería más profundamente para comprender completamente sus posibilidades.

10.2.1. Instalación de ZThreads

Por favor, note que la librería ZThread es un proyecto independiente y no está soportada por el autor de este libro; simplemente estamos usando la librería en este capítulo y no podemos dar soporte técnico a las características de la instalación. Mira el sitio web de ZThread para obtener soporte en la instalación y reporte de errores.

La librería ZThread se distribuye como código fuente. Después de descargarla (versión 2.3 o superior) desde la web de ZThread, debes compilar la librería primero, y después configurar tu proyecto para que use la librería. -->

El método habitual para compilar la librería ZThreads para los distintos sabores de UNIX (Linux, SunOS, Cygwin, etc) es usar un script de configuración. Después de desempaquetar los archivos (usando tar), simplemente ejecuta: ./configure && make install

en el directorio principal de ZThreads para compilar e instalar una copia de la librería en directorio /usr/local. Puedes personalizar algunas opciones cuando uses el script, incluída la localización de los ficheros. Para más detalles, utiliza este comando: ./configure ?help

El código de ZThreads está estructurado para simplificar la compilación para otras plataformas (como Borland, Microsoft y Metrowerks). Para hacer esto, crea un nuevo proyecto y añade todos los archivos .cxx en el directorio src de ZThreads a

la lista de archivos a compilar. Además, asegúrate de incluir el directorio incluído del archivo en la ruta de búsqueda de la cabecera para tu proyecto???. Los detalles exactos variarán de compilador en compilador, por lo que necesitarás estar algo familiarizado con tu conjunto de herramientas para ser capaz de utilizar esta opción.

Una vez la compilación ha finalizado con éxito, el siguiente paso es crear un proyecto que use la nueva librería compilada. Primero, permite al compilador saber donde están localizadas las cabeceras, por lo que tu instrucción #include funcionará correctamente. Habitualmente, necesitarás en tu proyecto una opción como se muestra:

```
-I/path/to/installation/include
```

Si utilizaste el script de configuración, la ruta de instalación será el prefijo de la que definiste (por defecto, /usr/local). Si utilizaste uno de los archivos de proyecto en la creación del directorio, la ruta instalación debería ser simplemente la ruta al directorio principal del archivo ZThreads.

Después, necesitarás añadir una opción a tu proyecto que permitirá al enlazador saber donde está la librería. Si usaste el script de configuración, se parecerá a lo siguiente:

```
-L/path/to/installation/lib ?lZThread
```

Si usaste uno de los archivos del proyecto proporcionados, será similar a:

```
-L/path/to/installation/Debug ZThread.lib
```

De nuevo, si usaste el script de configuración, la ruta de instalación s será el prefijo de la que definistes. Si su utilizaste un archivo del proyecto, la ruta será la misma que la del directorio principal de ZThreads.

Nota que si estás utilizando Linux, o Cygwin (www.cygwin.com) bajo Windows, no deberías necesitar modificar la ruta de include o de la librería; el proceso por defecto de instalación tendrá cuidado para hacerlo por ti, normalmente.

En GNU/Linux, es posible que necesites añadir lo siguiente a tu .bashrc para que el sistema pueda encontrar la el archivo de la librería compartida LibZThread-x.x.so.0 cuando ejecute programas de este capítulo.

```
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
```

(Asumiendo que utilizas el proceso de instalación por defecto y la librería compartida acaba en /user/local/lib/; en otro caso, cambia la ruta por tu localización.

10.2.2. Definición de tareas

Un hilo cumple con una tarea, por lo que necesitas un manera de describir esa tarea. La clase Runnable proporciona unas interfaces comunes a ejecutar para cualquier tarea. Aquí está el núcleo de las clase Runnable de ZThread, que la encontrarás en el archivo Runnable.h dentro del directorio incluído, después de instalar la librería ZThread:

```
class Runnable {
public:
  virtual void run() = 0;
  virtual ~Runnable() {}
};
```

Al hacerla una clase base abstracta, Runnable es fácilmente combinable con una clase básica u otras clases.

Para definir una tarea, simplemente hereda de la clase Runnable y sobreescribe run( ) para que la tarea haga lo que quieres.

Por ejecomplo, la tarea LiftOff siguiente muestra la cuenta atrás antes de despegar:

```
//: C11:LiftOff.h
// Demonstration of the Runnable interface.
#ifndef LIFTOFF_H
#define LIFTOFF_H
#include <iostream>
#include "zthread/Runnable.h"

class LiftOff : public ZThread::Runnable {
  int countDown;
  int id;
public:
  LiftOff(int count, int ident = 0) :
    countDown(count), id(ident) {}
  ~LiftOff() {
    std::cout << id << " completed" << std::endl;
  }
  void run() {
    while(countDown--)
      std::cout << id << ":" << countDown << std::endl;
    std::cout << "Liftoff!" << std::endl;
  }
};
#endif // LIFTOFF_H ///:~
```

El identificador id sirve como distinción entre multiples instancias de la tarea. Si sólo quieres hacer una instancia, debes utilizar el valor por defecto para identificarla. El destructor te permitirá ver que tarea está destruída correctamente.

En el siguiente ejemplo, las tareas de run( ) no están dirigidas por hilos separados; directamente es una simple llamada en main( ):

```
//: C11:NoThread.cpp
#include "LiftOff.h"

int main() {
  LiftOff launch(10);
  launch.run();
} ///:~
```

Cuando una clase deriva de Runnable, debe tene una función run( ), pero no tiene nada de especial - no produce ninguna habibilidad innata en el hilo.

Para llevar a cabo el funcionamiento de los hilos, debes utilizas la clase Thread.

10.3. Utilización de los hilos

Para controlar un objeto Runnable con un hilo, crea un objeto Thread separado y utiliza un pontero Runnable al constructor de Thread. Esto lleva a cabo la inicialización del hilo y, después, llama a run ( ) de Runnable como un hilo capaz de ser interrumpido. Manejando LiftOff con un hilo, el ejemplo siguiente muestra como cualquier tarea puede ser ejecutada en el contexto de cualquier otro hilo:

```
//: C11:BasicThreads.cpp
// The most basic use of the Thread class.
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    Thread t(new LiftOff(10));
    cout << "Waiting for LiftOff" << endl;
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Synchronization_Exception forma parte de la librería ZThread y la clase base para todas las excepciones de ZThread. Se lanzará si hay un error al crear o usar un hilo.

Un constructor de `Thread` sólo necesita un puntero a un objeto Runnable. Al crear un objeto `Thread` se efectuará la incialización necesaria del hilo y después se llamará a `Runnable::run()`

Puede añadir más hilos fácilmente para controlar más tareas. A continuación, puede ver cómo los hilos se ejecutan con algún otro:

```
//: C11:MoreBasicThreads.cpp
// Adding more threads.
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

int main() {
  const int SZ = 5;
  try {
    for(int i = 0; i < SZ; i++)
```

```
      Thread t(new LiftOff(10, i));
    cout << "Waiting for LiftOff" << endl;
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

El segundo argumento del constructor de `LiftOff` identifica cada tarea. Cuando ejecute el programa, verá que la ejecución de las distintas tareas se mezclan a medida que los hilos entran y salen de su ejecución. Este intercambio está controlado automáticamente por el planificador de hilos. Si tiene múltiples procesadores en su máquina, el planificador de hilos distribuirá los hilos entre los procesadores de forma transparente.

El bucle for puede parecer un poco extraño a priori ya que se crea localmente dentro del bucle for e inmediatamente después sale del ámbito y es destruído. Esto hace que parezca que el hilo propiamente dicho pueda perderse inmediatamente, pero puede ver por la salida que los hilos, en efecto, están en ejecución hasta su finalización. Cuando crea un objeto `Thread`, el hilo asociado se registra en el sistema de hilos, que lo mantiene vivo. A pesar de que el objeto `Thread` local se pierde, el hilo sigue vivo hasta que su tarea asociada termina. Aunque puede ser poco intuitivo desde el punto de vista de C++, el concepto de hilos es la excepción de la regla: un hilo crea un hilo de ejecución separado que persiste después de que la llamada a función finalice. Esta excepción se refleja en la persistencia del hilo subyacente después de que el objeto desaparezca.

10.3.1. Creación de interfaces de usuarios interactivas

Como se dijo anteriormente, uno de las motivaciones para usar hilos es crear interfaces de usuario interactivas. Aunque en este libro no cubriremos las interfaces gráficas de usuario, verá un ejemplo sencillo de una interfaz de usuario basada en consola.

El siguiente ejemplo lee líneas de un archivo y las imprime a la consola, durmiéndose (suspender el hilo actual) durante un segundo después de que cada línea sea mostrada. (Aprenderá más sobre el proceso de dormir hilos en el capítulo.) Durante este proceso, el programa no busca la entrada del usuario, por lo que la IU no es interactiva:

```
//: C11:UnresponsiveUI.cpp {RunByHand}
// Lack of threading produces an unresponsive UI.
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace std;
using namespace ZThread;

int main() {
  cout << "Press <Enter> to quit:" << endl;
  ifstream file("UnresponsiveUI.cpp");
  string line;
  while(getline(file, line)) {
```

```
    cout << line << endl;
    Thread::sleep(1000); // Time in milliseconds
  }
  // Read input from the console
  cin.get();
  cout << "Shutting down..." << endl;
} ///:~
```

Para hacer este programa interactivo, puede ejecutar una tarea que muestre el archivo en un hilo separado. De esta forma, el hilo principal puede leer la entrada del usuario, por lo que el programa se vuelve interactivo:

```
//: C11:ResponsiveUI.cpp {RunByHand}
// Threading for a responsive user interface.
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class DisplayTask : public Runnable {
  ifstream in;
  string line;
  bool quitFlag;
public:
  DisplayTask(const string& file) : quitFlag(false) {
    in.open(file.c_str());
  }
  ~DisplayTask() { in.close(); }
  void run() {
    while(getline(in, line) && !quitFlag) {
      cout << line << endl;
      Thread::sleep(1000);
    }
  }
  void quit() { quitFlag = true; }
};

int main() {
  try {
    cout << "Press <Enter> to quit:" << endl;
    DisplayTask* dt = new DisplayTask("ResponsiveUI.cpp");
    Thread t(dt);
    cin.get();
    dt->quit();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
  cout << "Shutting down..." << endl;
} ///:~
```

Ahora el hilo main() puede responder inmediatamente cuando pulse Return e invocar quit() sobre `DisplayTask`.

Este ejemplo también muestra la necesidad de una comunicación entre tareas - la tarea en el hilo main() necesita parar al DisplayTask. Dado que tenemos un puntero a DisplayTask, puede pensar que bastaría con llamar al destructor de ese puntero para matar la tarea, pero esto hace que los programas sean poco fiables. El problema es que la tarea podría estar en mitad de algo importante cuando lo destruye y, por lo tanto, es probable que ponga el programa en un estado inestable. En este sentido, la propia tarea decide cuando es seguro terminar. La manera más sencilla de hacer esto es simplemente notificar a la tarea que desea detener mediante una bandera booleana. Cuando la tarea se encuentre en un punto estable puede consultar esa bandera y hacer lo que sea necesario para limpiar el estado después de regresar de run(). Cuando la tarea vuelve de run(), Thread sabe que la tarea se ha completado.

Aunque este programa es lo suficientemente simple para que no haya problemas, hay algunos pequeños defectos respecto a la comunicación entre pilas. Es un tema importante que se cubrirá más tarde en este capítulo.

10.3.2. Simplificación con Ejecutores

Utilizando los Ejecutores de ZThread, puede simplificar su código. Los Ejecutores proporcionan una capa de indirección entre un cliente y la ejecución de una tarea; a diferencia de un cliente que ejecuta una tarea directamente, un objeto intermediario ejecuta la tarea.

Podemos verlo utilizando un Ejecutor en vez de la creación explícita de objetos Thread en MoreBasicThreads.cpp. Un objeto LiftOff conoce cómo ejecutar una tarea específica; como el patrón Command, expone una única función a ejecutar. Un objeto Executor conoce como construir el contexto apropiado para lanzar objetos Runnable. En el siguiente ejemplo, ThreadedExecutor crea un hilo por tarea:

```
//: c11:ThreadedExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/ThreadedExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Note que algunos casos un Executor individual puede ser usado para crear y gestionar todo los hilos en su sistema. Debe colocar el código correspondiente a los hilos dentro de un bloque try porque el método execute() de un Executor puede lanzar una Synchronization_Exception si algo va mal. Esto es válido para cualquier función que implique cambiar el estado de un objeto de sincronización (arranque de hilos, la adquisición de mutexes, esperas en condiciones, etc.), tal y como aprenderá más adelante en este capítulo.

El programa finalizará cuando todas las tareas en el `Executor` hayan concluido.

En el siguiente ejemplo, `ThreadedExecutor` crea un hilo para cada tarea que quiera ejecutar, pero puede cambiar fácilmente la forma en la que esas tareas son ejecutadas reemplazando el `ThreadedExecutor` por un tipo diferente de `Executor`. En este capítulo, usar un `ThreadedExecutor` está bien, pero para código en producción puede resultar excesivamente costoso para la creación de muchos hilos. En ese caso, puede reemplazarlo por un `PoolExecutor`, que utilizará un conjunto limitado de hilos para lanzar las tareas registradas en paralelo:

```
//: C11:PoolExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/PoolExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    // Constructor argument is minimum number of threads:
    PoolExecutor executor(5);
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Con `PoolExecutor` puede realizar una asignación inicial de hilos costosa de una sola vez, por adelantado, y los hilos se reutilizan cuando sea posible. Esto ahorra tiempo porque no está pagando el gasto de la creación de hilos por cada tarea individual de forma constante. Además, en un sistema dirigido por eventos, los eventos que requieren hilos para manejarlos puede ser generados tan rápido como quiera, basta con traerlos del pool. No excederá los recursos disponibles porque `PoolExecutor` utiliza un número limitado de objetos `Thread`. Así, aunque en este libro se utilizará `ThreadedExecutors`, tenga en cuenta utilizar `PoolExecutor` para código en producción.

`ConcurrentExecutor` es como `PoolExecutor` pero con un tamaño fijo de hilos. Es útil para cualquier cosa que quiera lanzar en otro hilo de forma continua (una tarea de larga duración), como una tare que escucha conexiones entrantes en un socket. También es útil para tareas cortas que quiera lanzar en un hilo, por ejemplo, pequeñas tareas que actualizar un log local o remoto, o para un hilo que atienda a eventos.

Si hay más de una tarea registrada en un `ConcurrentExecutor`, cada una de ellas se ejecutará completamente hasta que la siguiente empiece; todas utilizando el mismo hilo. En el ejemplo siguiente, verá que cada tarea se completa, en el orden en el que fue registrada, antes de que la siguiente comience. De esta forma, un `ConcurrentExecutor` serializa las tareas que le fueron asignadas.

```
//: C11:ConcurrentExecutor.cpp
//{L} ZThread
#include <iostream>
```

```
#include "zthread/ConcurrentExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    ConcurrentExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Como un ConcurrentExecutor, un SynchronousExecutor se usa cuando quiera una única tarea se ejecute al mismo tiempo, en serie en lugar de concurrente. A diferencia de ConcurrentExecutor, un SynchronousExecutor no crea ni gestiona hilos sobre si mismo. Utiliza el hilo que añadió la tarea y, así, únicamente actúa como un punto focal para la sincronización. Si tiene n tareas registradas en un SynchronousExecutor, nunca habrá 2 tareas que se ejecuten a la vez. En lugar de eso, cada una se ejecutará hasta su finalización y la siguiente en la cola comenzará.

Por ejemplo, suponga que tiene un número de hilos ejecutando tareas que usan un sistema de archivos, pero está escribiendo código portable luego no quiere utilizar flock() u otra llamada al sistema operativo específica para bloquear un archivo. Puede lanzar esas tareas con un SynchronousExecutor para asegurar que solamente una de ellas, en un tiempo determinado, está ejecutándose desde cualquier hilo. De esta manera, no necesita preocuparse por la sincronización del recurso compartido (y, de paso, no se cargará el sistema de archivos). Una mejor solución pasa por sincronizar el recurso (lo cual aprenderá más adelante en este capítulo), sin embargo un SynchronousExecutor le permite evitar las molestias de obtener una coordinación adecuada para prototipar algo.

```
//: C11:SynchronousExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/SynchronousExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    SynchronousExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Cuando ejecuta el programa verá que las tareas son lanzadas en el orden en el que fueron registradas, y cada tarea se ejecuta completamente antes de que la siguiente

empiece. ¿Qué es lo que no ve y que hace que no se creen nuevos hilos? El hilo main() se usa para cada tarea, y debido a este ejemplo, ese es el hilo que registra todas las tareas. Podría no utilizar un `SynchronousExecutor` en código en producción porque, principalmente, es para prototipado.

10.3.3. Ceder el paso

Si sabe que ha logrado realizar lo que necesita durante una pasada a través de un bucle en su función run() (la mayoría de las funciones run() suponen un periodo largo de tiempo de ejecución), puede darle un toque al mecanismo de planificación de hilos, decirle que ya ha hecho suficiente y que algún otro hilo puede tener la CPU. Este toque (y es un toque - no hay garantía de que su implementación vaya a escucharlo) adopta la forma de la función yield().

Podemos construir una versión modificada de los ejemplos de `LiftOff` cediendo el paso después de cada bucle:

```
//: C11:YieldingTask.cpp
// Suggesting when to switch threads with yield().
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class YieldingTask : public Runnable {
  int countDown;
  int id;
public:
  YieldingTask(int ident = 0) : countDown(5), id(ident) {}
  ~YieldingTask() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const YieldingTask& yt) {
    return os << "#" << yt.id << ": " << yt.countDown;
  }
  void run() {
    while(true) {
      cout << *this << endl;
      if(--countDown == 0) return;
      Thread::yield();
    }
  }
};

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new YieldingTask(i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Puede ver que la tarea del método `run()` es en un bucle infinito en su totalidad. Utilizando yield(), la salida se equilibra bastante que en el caso en el que no se cede el paso. Pruebe a comentar la llamada a Thread::yield() para ver la diferencia. Sin embargo, en general, yield() es útil en raras ocasiones, y no puede contar con ella para realizar un afinamiento serio sobre su aplicación.

10.3.4. Dormido

Otra forma con la que puede tener control sobre el comportamiento de su hilos es llamando a sleep() para cesar la ejecución de uno de ellos durante un número de milisegundos dado. En el ejemplo que viene a continuación, si cambia la llamada a yield() por una a sleep(), obtendrá lo siguiente:

```
//: C11:SleepingTask.cpp
// Calling sleep() to pause for awhile.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class SleepingTask : public Runnable {
  int countDown;
  int id;
public:
  SleepingTask(int ident = 0) : countDown(5), id(ident) {}
  ~SleepingTask() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const SleepingTask& st) {
    return os << "#" << st.id << ": " << st.countDown;
  }
  void run() {
    while(true) {
      try {
        cout << *this << endl;
        if(--countDown == 0) return;
        Thread::sleep(100);
      } catch(Interrupted_Exception& e) {
        cerr << e.what() << endl;
      }
    }
  }
};

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new SleepingTask(i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
```

```
  }
} ///:~
```

Thread::sleep() puede lanzar una `Interrupted_Exception`(sobre las interrupciones aprenderá más adelante), y puede ver que esta excepción se captura en run(). Sin embargo, la tarea se crea y se ejecuta dentro de un bloque try en main() que captura `Interrupted_Exception` (la clase base para todas las excepciones de ZThread), por lo que ¿no sería posible ignorar la excepción en run() y asumir que se propagará al manejador en main()?. Esto no funcionará porque las excepciones no se propagarán a lo largo de los hilos para volver hacia main(). De esta forma, debe manejar cualquier excepción que pueda ocurrir dentro de una tarea de forma local.

Notará que los hilos tienden a ejecutarse en cualquier orden, lo que quiere decir que sleep() tampoco es una forma de controlar el orden de la ejecución de los hilos. Simplemente para la ejecución del hilo durante un rato. La única garantía que tiene es que el hilo se dormirá durante, al menos, 100 milisegundos (en este ejemplo), pero puede que tarde más después de que el hilo reinicie la ejecución ya que el planificador de hilos tiene que volver a él tras haber expirado el intervalo.

Si debe tener control sobre el orden de la ejecución de hilos, su mejor baza es el uso de controles de sincronización (descritos más adelante) o, en algunos casos, no usar hilos en todo, FIXMEbut instead to write your own cooperative routines that hand control to each other in a specified order.

10.3.5. Prioridad

La prioridad de un hilo representa la importancia de ese hilo para el planificador. Pese a que el orden en que la CPU ejecuta un conjunto de hilos es indeterminado, el planificador tenderá a ejecutar el hilo con mayor prioridad de los que estén esperando. Sin embargo, no quiere decir que hilos con menos prioridad no se ejecutarán (es decir, no tendrá bloqueo de un hilo a causa de las prioridades). Simplemente, los hilos con menos prioridad tenderán a ejecutarse menos frecuentemente.

Se ha modificado MoreBasicThreads.cpp para mostrar los niveles de prioridad. Las prioridades se ajustan utilizando la función setPriority() de `Thread`.

```
//: C11:SimplePriorities.cpp
// Shows the use of thread priorities.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double pi = 3.14159265358979323846;
const double e = 2.7182818284590452354;

class SimplePriorities : public Runnable {
  int countDown;
  volatile double d; // No optimization
  int id;
public:
  SimplePriorities(int ident=0): countDown(5), id(ident) {}
  ~SimplePriorities() {
    cout << id << " completed" << endl;
```

```
  }
  friend ostream&
  operator<<(ostream& os, const SimplePriorities& sp) {
    return os << "#" << sp.id << " priority: "
      << Thread().getPriority()
      << " count: "<< sp.countDown;
  }
  void run() {
    while(true) {
      // An expensive, interruptable operation:
      for(int i = 1; i < 100000; i++)
        d = d + (pi + e) / double(i);
      cout << *this << endl;
      if(--countDown == 0) return;
    }
  }
};

int main() {
  try {
    Thread high(new SimplePriorities);
    high.setPriority(High);
    for(int i = 0; i < 5; i++) {
      Thread low(new SimplePriorities(i));
      low.setPriority(Low);
    }
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

En este ejemplo, el operador <<() se sobreescribe para mostrar el identificador, la prioridad y el valor de countDown de la tarea.

Puede ver que el nivel de prioridad del hilo es el más alto, y que el resto de hilos tienen el nivel más bajo. No utilizamos `Executor` en este ejemplo porque necesitamos acceder directamente al hilo para configurar sus propiedades.

Dentro de SimplePriorities::run() se ejecutan 100,000 veces un costoso conjunto de cálculos en punto flotante. La variable d es volátil para intentar garantizar que ningún compilador hace optimizaciones. Sin este cálculo, no comprobará el efecto de la configuración de los niveles de prioridad. (Pruébelo: comente el bucle for que contiene los cálculos en doble precisión.) Con el cálculo puede ver que el planificador de hilos al hilo high se le da más preferencia. (Al menos, este fue el comportamiento sobre una máquina Windows). El cálculo tarda lo suficiente para que el mecanismo de planificación de hilos lo salte, cambie hilos y tome en cuenta las prioridades para que el hilo high tenga preferencia.

También, puede leer la prioridad de un hilo existente con getPriority() y cambiarla en cualquier momento (no sólo antes de que el hilo se ejecute, como en SimplePriorities.cpp) con setPriority().

La correspondencia de las prioridades con el sistema operativo es un problema. Por ejemplo, Windows 2000 tiene siete niveles de prioridades, mientras que Solaris de Sun tiene 231. El único enfoque portable es ceñirse a los niveles discretos de prioridad, como Low, Medium y High utilizados en la librería ZThread.

10.4. Comparición de recursos limitados

Piense en un programa con un único hilo como una solitaria entidad moviéndose a lo largo del espacio de su problema y haciendo una cosa en cada instante. Debido a que sólo hay una entidad, no tiene que preocuparse por el problema de dos entidades intentando usar el mismo recurso al mismo tiempo: problemas como dos personas intentando aparcar en el mismo sitio, pasar por una puerta al mismo tiempo o incluso hablar al mismo tiempo.

Con multihilado las cosas ya no son solitarias, pero ahora tiene la posibilidad de tener dos o más hilos intentando utilizar un mismo recurso a la vez. Esto puede causar dos problemas distintos. El primero es que el recurso necesario podría no existir. En C++, el programador tiene un control total sobre la vida de los objetos, y es fácil crear hilos que intenten usar objetos que han sido destruidos antes de que esos hilos hayan finalizado.

El segundo problema es que dos o más hilos podrían chocar cuando intenten acceder al mismo dispositivo al mismo tiempo. Si no previene esta colisión, tendrá dos hilos intentando acceder a la misma cuenta bancaria al mismo tiempo, imprimir en la misma impresora, ajustar la misma válvula, etc.

Esta sección presenta el problema de los objetos que desaparecen mientras las tareas aún están usándolos y el problema del choque entre tareas sobre recursos compartidos. Aprenderá sobre las herramientas que se usan para solucionar esos problemas.

10.4.1. Aseguramiento de la existencia de objetos

La gestión de memoria y recursos son las principales preocupaciones en C++. Cuando crea cualquier programa en C++, tiene la opción de crear objetos en la pila o en el heap (utilizando new). En un programa con un solo hilo, normalmente es sencillo seguir la vida de los objetos con el fin de que no tenga que utilizar objetos que ya están destruidos.

Los ejemplos mostrados en este capítulo crean objetos `Runnable` en el heap utilizando new, se dará cuenta que esos objetos nunca son destruidos explícitamente. Sin embargo, podrá por la salida cuando ejecuta el programa que la biblioteca de hilos sigue la pista a cada tarea y, eventualmente, las destruye. Esto ocurre cuando el método Runnable::run() finaliza - volver de run() indica que la tarea ha finalizado.

Recargar el hilo al destruir una tarea es un problema. Ese hilo sabe necesariamente si otro necesita hacer referencia a ese Runnable, y por ello el Runnable podría ser destruido prematuramente. Para ocuparse de este problema, el mecanismo de la biblioteca ZThread mantiene un conteo de referencias sobre las tareas. Una tarea se mantiene viva hasta que su contador de referencias se pone a cero, en este punto la tarea se destruye. Esto quiere decir que las tareas tienen que ser destruidas dinámicamente siempre, por lo que no pueden ser creadas en la pila. En vez de eso, las tareas deben ser creadas utilizando new, tal y como puede ver en todos los ejemplos de este capítulo. tareas in ZThreads

Además, también debe asegurar que los objetos que no son tareas estarán vivos tanto tiempo como el que las tareas necesiten de ellos. Por otro lado, resulta sencillo para los objetos utilizados por las tareas salir del ámbito antes de que las tareas hayan concluido. Si ocurre esto, las tareas intentarán acceder zonas de almacenamiento ilegales y provocará que el programa falle. He aquí un simple ejemplo:

```
//: C11:Incrementer.cpp {RunByHand}
// Destroying objects while threads are still
// running will cause serious problems.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Count {
  enum { SZ = 100 };
  int n[SZ];
public:
  void increment() {
    for(int i = 0; i < SZ; i++)
      n[i]++;
  }
};

class Incrementer : public Runnable {
  Count* count;
public:
  Incrementer(Count* c) : count(c) {}
  void run() {
    for(int n = 100; n > 0; n--) {
      Thread::sleep(250);
      count->increment();
    }
  }
};

int main() {
  cout << "This will cause a segmentation fault!" << endl;
  Count count;
  try {
    Thread t0(new Incrementer(&count));
    Thread t1(new Incrementer(&count));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Podría parecer a priori que la clase `Count` es excesiva, pero si únicamente n es un int (en lugar de una matriz), el compilador puede ponerlo dentro de un registro y ese almacenamiento seguirá estando disponible (aunque técnicamente es ilegal) después de que el objeto `Count` salga del ámbito. Es difícil detectar la violación de memoria en este caso. Sus resultados podrían variar dependiendo de su compilador y de su sistema operativo, pero pruebe a que n sea un int y verá qué ocurre. En cualquier evento, si `Count` contiene una matriz de ints y como antes, el compilador está obligado a ponerlo en la pila y no en un registro.

`Incrementer` es una tarea sencilla que utiliza un objeto `Count`. En main(), puede ver que las tareas `Incrementer` se ejecutan el tiempo suficiente para que el salga del ámbito, por lo que la tarea intentará acceder a un objeto que no existe. Esto produce un fallo en el programa. objeto `Count`

Para solucionar este problema, debemos garantizar que cualquiera de los objetos compartidos entre tareas estarán accesibles tanto tiempo como las tareas los necesiten. (Si los objetos no fueran compartidos, podrían estar directamente dentro de las clases de las tareas y, así, unir su tiempo de vida a la tarea.) Dado que no queremos que el propio ámbito estático del programa controle el tiempo de vida del objeto, pondremos el en heap. Y para asegurar que el objeto no se destruye hasta que no haya objetos (tareas, en este caso) que lo estén utilizando, utilizaremos el conteo de referencias.

El conteo de referencias se ha explicado a lo largo del volumen uno de este libro y además se revisará en este volumen. La librería ZThread incluye una plantilla llamada `CountedPtr` que automáticamente realiza el conteo de referencias y destruye un objeto cuando su contador de referencias vale cero. A continuación, se ha modificado el programa para que utilice `CountedPtr` para evitar el fallo:

```
//: C11:ReferenceCounting.cpp
// A CountedPtr prevents too-early destruction.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count {
  enum { SZ = 100 };
  int n[SZ];
public:
  void increment() {
    for(int i = 0; i < SZ; i++)
      n[i]++;
  }
};

class Incrementer : public Runnable {
  CountedPtr<Count> count;
public:
  Incrementer(const CountedPtr<Count>& c ) : count(c) {}
  void run() {
    for(int n = 100; n > 0; n--) {
      Thread::sleep(250);
      count->increment();
    }
  }
};

int main() {
  CountedPtr<Count> count(new Count);
  try {
    Thread t0(new Incrementer(count));
    Thread t1(new Incrementer(count));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Ahora Incrementer contiene un objeto CountedPtr, que gestiona un Count. En la función main(), los objetos CountedPtr se pasan a los dos objetos Incrementer por valor, por lo que se llama el constructor de copia, incrementando el conteo de referencias. Mientras la tarea esté ejecutándose, el contador de referencias no valdrá cero, por lo que el objeto Count utilizado por CountedPtr no será destruído. Solamente cuando todas las tareas que utilice el Count terminen se llamará al destructor (automáticamente) sobre el objeto Count por el CountedPtr.

Siempre que tenga una tarea que utilice más de un objeto, casi siempre necesitará controlar aquellos objetos utilizando la plantilla CountedPtr para evitar problemas derivados del tiempo de vida de los objetos.

10.4.2. Acceso no apropiado a recursos

Considere el siguiente ejemplo, donde una tarea genera números constantes y otras tareas consumen esos números. Ahora, el único trabajo de los hilos consumidores es probar la validez de los números constantes.

Primeramente, definiremos EvenChecker, el hilo consumidor, puesto que será reutilizado en todos los ejemplos siguientes. Para desacoplar EvenChecker de los varios tipos de generadores con los que experimentaremos, crearemos una interfaz llamada Generator que contiene el número mínimo de funciones que EvenChecker necesita conocer: por lo que tiene una función nextValue() y que puede ser cancelada.

```
//: C11:EvenChecker.h
#ifndef EVENCHECKER_H
#define EVENCHECKER_H
#include <iostream>
#include "zthread/CountedPtr.h"
#include "zthread/Thread.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadedExecutor.h"

class Generator : public ZThread::Cancelable {
  bool canceled;
public:
  Generator() : canceled(false) {}
  virtual int nextValue() = 0;
  void cancel() { canceled = true; }
  bool isCanceled() { return canceled; }
};

class EvenChecker : public ZThread::Runnable {
  ZThread::CountedPtr<Generator> generator;
  int id;
public:
  EvenChecker(ZThread::CountedPtr<Generator>& g, int ident)
  : generator(g), id(ident) {}
  ~EvenChecker() {
    std::cout << "~EvenChecker " << id << std::endl;
  }
  void run() {
    while(!generator->isCanceled()) {
      int val = generator->nextValue();
      if(val % 2 != 0) {
```

```
        std::cout << val << " not even!" << std::endl;
        generator->cancel(); // Cancels all EvenCheckers
      }
    }
  }
  // Test any type of generator:
  template<typename GenType> static void test(int n = 10) {
    std::cout << "Press Control-C to exit" << std::endl;
    try {
      ZThread::ThreadedExecutor executor;
      ZThread::CountedPtr<Generator> gp(new GenType);
      for(int i = 0; i < n; i++)
        executor.execute(new EvenChecker(gp, i));
    } catch(ZThread::Synchronization_Exception& e) {
      std::cerr << e.what() << std::endl;
    }
  }
};
#endif // EVENCHECKER_H ///:~
```

La clase `Generator` presenta la clase abstracta `Cancelable`, que es parte de la biblioteca de ZThread. El propósito de `Cancelable` es proporcionar una interfaz consistente para cambiar el estado de un objeto via cancel() y ver si el objeto ha sido cancelado con la función isCanceled(). Aquí utilizamos el enfoque simple de una bandera de cancelación booleana similar a quitFlag, vista previamente en ResponsiveUI.cpp. Note que en este ejemplo la clase que es `Cancelable` no es `Runnable`. En su lugar, toda tarea `EvenChecker` que dependa de un objeto `Cancelable` (el `Generator`) lo comprueba para ver que ha sido cancelado, como puede ver en run(). De esta manera, las tareas que comparten recursos comunes (el `Cancelable Generator`) estén atentos a la señal de ese recurso para terminar. Esto elimina la también conocida condición de carrera, donde dos o más tareas compiten por responder una condición y, así, colisionar o producir resultados inconsistentes. Debe pensar sobre esto cuidadosamente y protegerse de todas las formas posible de los fallos de un sistema concurrente. Por ejemplo, una tarea no puede depender de otra porque el orden de finalización de las tareas no está garantizado. En este sentido, eliminamos la potencial condición de carrera haciendo que las tareas dependan de objetos que no son tareas (que son contados referencialmente utilizando `CountedPtr`.

En las secciones posteriores, verá que la librería ZThread contiene más mecanismos generales para la terminación de hilos.

Debido a que muchos objetos `EvenChecker` podrían terminar compartiendo un `Generator`, la plantilla `CountedPtr` se usa para contar las referencias de los objetos `Generator`.

El último método en `EvenChecker` es un miembro estático de la plantilla que configura y realiza una comprobación de los tipos de `Generator` creando un `CountedPtr` dentro y, seguidamente, lanzar un número de `EvenChecker`s que usan ese `Generator`. Si el `Generator` provoca un fallo, test() lo reportará y volverá; en otro caso, deberá pulsar Control-C para finalizarlo.

Las tareas `EvenChecker` leen constantemente y comprueban que los valores de sus `Generator`s asociados. Vea que si generator->isCanceled() es verdadero, run() retorna, con lo que se le dice al `Executor` de EvenChecker::test() que la tarea se ha completado. Cualquier tarea `EvenChecker` puede llamar a cancel() sobre su `Generator` asociado, lo que causará que todos los demás `EvenChecker`s que utilicen

ese Generator finalicen con elegancia.

EvenGenerator es simple - nextValue() produce el siguiente valor constante:

```
//: C11:EvenGenerator.cpp
// When threads collide.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class EvenGenerator : public Generator {
  unsigned int currentEvenValue; // Unsigned can't overflow
public:
  EvenGenerator() { currentEvenValue = 0; }
  ~EvenGenerator() { cout << "~EvenGenerator" << endl; }
  int nextValue() {
    ++currentEvenValue; // Danger point here!
    ++currentEvenValue;
    return currentEvenValue;
  }
};

int main() {
  EvenChecker::test<EvenGenerator>();
} ///:~
```

Es posible que un hilo llame a nextValue() después de el primer incremento de currentEvenValue y antes del segundo (en el lugar "Danger point here!" del código comentado), que pone el valor en un estado "incorrecto". Para probar que esto puede ocurrir, EventChecker::test() crea un grupo de objetos EventChecker para leer continuamente la salida de un EvenGenerator y ver si cada valor es constante. Si no es así, el error se reporta y el programa finaliza.

Este programa podría no detectar el problema hasta que EvenGenerator ha completado varios ciclos, dependiendo de las particuliaridades de su sistema operativo y otros detalles de implementación. Si quiere ver que falla mucho más rápido, pruebe a poner una llamada a yield() entre el primero y segundo incremento. En algún evento, fallará puntualmente a causa de que los hilos EvenChecker pueden acceder a la información en EvenGenerator mientras se encuentra en un estado "incorrecto".

10.4.3. Control de acceso

En el ejemplo anterior se muestra el problema fundamental a la hora de utilizar hilos: nunca sabrá cuándo un hilo puede ser ejecutado. Imagínese sentado en la mesa con un tenedor, a punto de coger el último pedazo de comida de un plato y tan pronto como su tenedor lo alcanza, de repente, la comida se desvanece (debido a que su hilo fue suspendido y otro vino y se comió la comida). Ese es el problema que estamos tratando a la hora de escribir programas concurrentes.

En ocasiones no le importará si un recurso está siendo accedido a la vez que intenta usarlo. Pero en la mayoría de los casos sí, y para trabajar con múltiples hilos

necesitará alguna forma de evitar que dos hilos accedan al mismo recurso, al menos durante períodos críticos.

Para prevenir este tipo de colisiones existe una manera sencilla que consiste en poner un bloqueo sobre un recursos cuando un hilo trata de usarlo. El primer hilo que accede al recursos lo bloquea y, así, otro hilo no puede acceder al recurso hasta que no sea desbloqueado, momento en el que este hilo lo vuelve a bloquear y lo vuelve a usar, y así sucesivamente.

De esta forma, tenemos que ser capaces de evitar cualquier tarea de acceso a memoria mientras ese almacenamiento no esté en un estado adecuado. Esto es, necesitamos tener un mecanismo que excluya una segunda tarea sobre el acceso a memoria cuando una primera tarea ya está usándola. Esta idea es fundamental para todo sistema multihilado y se conoce como exclusión mutua; abreviado como mutex. La biblioteca ZThread tiene un mecanismo de mutex en el fichero de cabecera Mutex.h.

Para solucionar el problema en el programa anterior, identificaremos las secciones críticas donde debe aplicarse la exclusión mutua; posteriormente, antes de entrar en la sección crítica adquiriremos el mutex y lo liberaremos cuando finalice la sección crítica. Únicamente un hilo podrá adquirir el mutex al mismo tiempo, por lo que se logra exclusión mutua:

```
//: C11:MutexEvenGenerator.cpp {RunByHand}
// Preventing thread collisions with mutexes.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
using namespace ZThread;
using namespace std;

class MutexEvenGenerator : public Generator {
  unsigned int currentEvenValue;
  Mutex lock;
public:
  MutexEvenGenerator() { currentEvenValue = 0; }
  ~MutexEvenGenerator() {
    cout << "~MutexEvenGenerator" << endl;
  }
  int nextValue() {
    lock.acquire();
    ++currentEvenValue;
    Thread::yield(); // Cause failure faster
    ++currentEvenValue;
    int rval = currentEvenValue;
    lock.release();
    return rval;
  }
};

int main() {
  EvenChecker::test<MutexEvenGenerator>();
} ///:~
```

`MutexEvenGenerator` añade un `Mutex` llamado lock y utiliza acquire() y re-

lease() para crear una sección crítica con nextValue(). Además, se ha insertado una llamada a Thread::yield() entre los dos incrementos, para aumentar la probabilidad de que haya un cambio de contexto mientras currentEvenValue se encuentra en un estado extraño. Este hecho no producirá un fallo ya que el mutex evita que más de un hilo esté en la sección crítica al mismo tiempo, pero llamar a yield() es una buena forma de provocar un fallo si este ocurriera.

Note que nextValue() debe capturar el valor de retorno dentro de la sección crítica porque si lo devolviera dentro de la sección critica no liberaría lock y así evitar que fuera adquirido. (Normalmente, esto conllevaría un interbloqueo, de lo cual aprenderá sobre ello al final de este capítulo.)

El primer hilo que entre en nextValue() adquirirá lock y cualquier otro hilo que intente adquirirlo será bloqueado hasta que el primer hilo libere lock. En ese momento, el mecanismo de planificación selecciona otro hilo que esté esperando en lock. De esta manera, solo un hilo puede pasar a través del código custodiado por el mutex al mismo tiempo.

10.4.4. Código simplificado mediante guardas

El uso de mutexes se convierte rápidamente complicado cuando se introducen excepciones. Para estar seguro de que el mutes siempre se libera, debe asegurar que cualquier camino a una excepción incluya una llamada a release(). Además, cualquier función que tenga múltiples caminos para retornar debe asegurar cuidadosamente que se llama a release() en el momento adecuado.

Esos problemas pueden se fácilmente solucionados utilizando el hecho de que los objetos de la pila (automáticos) tiene un destructor que siempre se llama sea cual sea la forma en que salga del ámbito de la función. En la librería ZThread, esto se implementa en la plantilla `Guard`. La plantilla `Guard` crea objetos que adquieren un objeto `Lockable` cuando se construyen y lo liberan cuando son destruidos. Los objetos `Guard` creados en la pila local serán eliminados automáticamente independientemente de la forma en el que la función finalice y siempre desbloqueará el objeto `Lockable`. A continuación, el ejemplo anterior reimplementado para utilizar `Guard`s:

```
//: C11:GuardedEvenGenerator.cpp {RunByHand}
// Simplifying mutexes with the Guard template.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;
using namespace std;

class GuardedEvenGenerator : public Generator {
  unsigned int currentEvenValue;
  Mutex lock;
public:
  GuardedEvenGenerator() { currentEvenValue = 0; }
  ~GuardedEvenGenerator() {
    cout << "~GuardedEvenGenerator" << endl;
  }
  int nextValue() {
```

```
    Guard<Mutex> g(lock);
    ++currentEvenValue;
    Thread::yield();
    ++currentEvenValue;
    return currentEvenValue;
  }
};

int main() {
  EvenChecker::test<GuardedEvenGenerator>();
} ///:~
```

Note que el valor de retorno temporal ya no es necesario en nextValue(). En general, hay menos código que escribir y la probabilidad de errores por parte del usuario se reduce en gran medida.

Una característica interesante de la plantilla `Guard` es que puede ser usada para manipular otros elementos de seguridad. Por ejemplo, un segundo `Guard` puede ser utilizado temporalmente para desbloquear un elemento de seguridad:

```
//: C11:TemporaryUnlocking.cpp
// Temporarily unlocking another guard.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;

class TemporaryUnlocking {
  Mutex lock;
public:
  void f() {
    Guard<Mutex> g(lock);
    // lock is acquired
    // ...
    {
      Guard<Mutex, UnlockedScope> h(g);
      // lock is released
      // ...
      // lock is acquired
    }
    // ...
    // lock is released
  }
};

int main() {
  TemporaryUnlocking t;
  t.f();
} ///:~
```

Un `Guard` también puede utilizarse para adquirir un lock durante un determinado tiempo y, después, liberarlo:

```
//: C11:TimedLocking.cpp
```

```
// Limited time locking.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;

class TimedLocking {
  Mutex lock;
public:
  void f() {
    Guard<Mutex, TimedLockedScope<500> > g(lock);
    // ...
  }
};

int main() {
  TimedLocking t;
  t.f();
} ///:~
```

En este ejemplo, se lanzará una `Timeout_Exception` si el lock no puede ser adquirido en 500 milisegundos.

Sincronización de clases completas

La librería ZThread también proporciona la plantilla `GuardedClass` para crear automáticamente un recubrimiento de sincronización para toda una clase. Esto quiere decir que cualquier método de una clase estará automáticamente protegido:

```
//: C11:SynchronizedClass.cpp {-dmc}
//{L} ZThread
#include "zthread/GuardedClass.h"
using namespace ZThread;

class MyClass {
public:
  void func1() {}
  void func2() {}
};

int main() {
  MyClass a;
  a.func1(); // Not synchronized
  a.func2(); // Not synchronized
  GuardedClass<MyClass> b(new MyClass);
  // Synchronized calls, only one thread at a time allowed:
  b->func1();
  b->func2();
} ///:~
```

El objeto a no está sincronizado, por lo que func1() y func2() pueden ser llamadas en cualquier momento por cualquier número de hilos. El objeto b está protegido por el recubrimiento `GuardedClass`, así que cada método se sincroniza automáticamente y solo se puede llamar a una función por objeto en cualquier instante.

El recubrimiento bloquea un tipo de nivel de granularidad, que podría afectar al rendimiento.[151] Si una clase contiene funciones no vinculadas, puede ser mejor sincronizarlas internamente con 2 locks diferentes. Sin embargo, si se encuentra haciendo esto, significa que la clase contiene grupos de datos que puede no estar fuertemente asociados. Considere dividir la clase en dos.

Proteger todos los métodos de una clase con un mutex no hace que esa clase sea segura automáticamente cuando se utilicen hilos. Debe tener cuidado con estas cuestiones para garantizar la seguridad cuando se usan hilos.

10.4.5. Almacenamiento local al hilo

Una segunda forma de eliminar el problema de colisión de tareas sobre recursos compartidos es la eliminación de las variables compartidas, lo cual puede realizarse mediante la creación de diferentes almacenamientos para la misma variable, uno por cada hilo que use el objeto. De esta forma, si tiene cinco hilos que usan un objeto con una variable x, el almacenamiento local al hilo genera automáticamente cinco porciones de memoria distintas para almacenar x. Afortunadamente, la creación y gestión del almacenamiento local al hilo la lleva a cabo una plantilla de ZThread llamada `ThreadLocal`, tal y como se puede ver aquí:

```
//: C11:ThreadLocalVariables.cpp {RunByHand}
// Automatically giving each thread its own storage.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadLocal.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class ThreadLocalVariables : public Cancelable {
  ThreadLocal<int> value;
  bool canceled;
  Mutex lock;
public:
  ThreadLocalVariables() : canceled(false) {
    value.set(0);
  }
  void increment() { value.set(value.get() + 1); }
  int get() { return value.get(); }
  void cancel() {
    Guard<Mutex> g(lock);
    canceled = true;
  }
  bool isCanceled() {
    Guard<Mutex> g(lock);
    return canceled;
  }
};

class Accessor : public Runnable {
```

```
  int id;
  CountedPtr<ThreadLocalVariables> tlv;
public:
  Accessor(CountedPtr<ThreadLocalVariables>& tl, int idn)
  : id(idn), tlv(tl) {}
  void run() {
    while(!tlv->isCanceled()) {
      tlv->increment();
      cout << *this << endl;
    }
  }
  friend ostream&
    operator<<(ostream& os, Accessor& a) {
    return os << "#" << a.id << ": " << a.tlv->get();
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CountedPtr<ThreadLocalVariables>
      tlv(new ThreadLocalVariables);
    const int SZ = 5;
    ThreadedExecutor executor;
    for(int i = 0; i < SZ; i++)
      executor.execute(new Accessor(tlv, i));
    cin.get();
    tlv->cancel(); // All Accessors will quit
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Cuando crea un objeto `ThreadLocal` instanciando la plantilla, únicamente puede acceder al contenido del objeto utilizando los métodos set() y get(). El método get() devuelve una copia del objeto que está asociado a ese hilo, y set() inserta su argumento dentro del objeto almacenado para ese hilo, devolviendo el objeto antiguo que se encontraba almacenado. Puede comprobar que esto se utiliza en increment() y get() de `ThreadLocalVariables`.

Ya que tlv se comparte en múltiples objetos `Accessor`, está escrito como un `Cancelable`, por lo que los `Accessors` puede recibir señales cuando queramos parar el sistema.

Cuando ejecute este programa se evidenciará que se reserva para cada hilo su propio almacenamiento.

10.5. Finalización de tareas

En los ejemplos anteriores, hemos visto el uso de una "bandera de terminación" o de la interfaz `Cancelable` para finalizar una tarea. Este es un enfoque razonable para el problema. Sin embargo, en algunas ocasiones la tarea tiene que ser finalizada más abruptamente. En esta sección, aprenderá sobre las cuestiones y problemas de este tipo de finalización.

Primeramente, veamos en un ejemplo que no sólo demuestra el problema de la

finalización sino que, además, es un ejemplo adicional de comparición de recursos. Para mostrar el ejemplo, primero necesitaremos resolver el problema de la colisión de iostream.

10.5.1. Prevención de coliciones en iostream

FIXME: dos versiones: Podría haberse dado cuenta en los anteriores ejemplos que la salida es confusa en algunas ocasiones. Los iostreams de C++ no fueron creados pensando en el sistema de hilos, por lo que no hay Puede haberse dado cuenta en los ejemplos anteriores que la salida es FIXMEconfusa. El sistema iostream de C++ no fue creado con el sistema de hilos en mente, por no lo que no hay nada que prevenga que la salida de un hilo interfiera con la salida de otro hilo. Por ello, debe escribir aplicaciones de tal forma que sincronicen el uso de iostreams.

Para solucionar el problema, primero necesitamos crear un paquete completo de salida y, después, decidir explícitamente cuando intentamos mandarlo a la consola. Una sencilla solución pasa por escribir la informacion en un ostringstream y posteriormente utilizar un único objeto con un mutex como punto de salida de todos los hilos, para evitar que más de un hilo escriba al mismo tiempo:

```
//: C11:Display.h
// Prevents ostream collisions.
#ifndef DISPLAY_H
#define DISPLAY_H
#include <iostream>
#include <sstream>
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

class Display { // Share one of these among all threads
  ZThread::Mutex iolock;
public:
  void output(std::ostringstream& os) {
    ZThread::Guard<ZThread::Mutex> g(iolock);
    std::cout << os.str();
  }
};
#endif // DISPLAY_H ///:~
```

De esta manera, predefinimos la función estandar `operator<<()` y el objeto puede ser construido en memoria utilizando operadores habituales de ostream. Cuando una tarea quiere mostrar una salida, crea un objeto ostringstream temporal que utiliza FIXME. Cuando llama a output(), el mutex evita que varios hilos escriban a este objeto `Display`. (Debe usar solo un objeto `Display` en su programa, tal y como verá en los siguientes ejemplos.)

Todo esto muestra la idea básica pero, si es necesario, puede construir un entorno más elaborado. Por ejemplo, podría forzar el requisito de que solo haya un objeto `Display` en un programa haciéndolo Singleton. (La librería ZThread tiene una plantilla `Singleton` para dar soporte a Singletons).

10.5.2. El jardín ornamental

En esta simulación, al comité del jardín le gustaría saber cuanta gente entra en el jardín cada día a través de distintas puertas. Cada puerta tiene un FIXMEturnstile o algún otro tipo de contador, y después de que el contador FIXMEturnstile se incrementa, aumenta una cuenta compartida que representa el número total de gente en el jardín.

```
//: C11:OrnamentalGarden.cpp {RunByHand}
//{L} ZThread
#include <vector>
#include <cstdlib>
#include <ctime>
#include "Display.h"
#include "zthread/Thread.h"
#include "zthread/FastMutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count : public Cancelable {
  FastMutex lock;
  int count;
  bool paused, canceled;
public:
  Count() : count(0), paused(false), canceled(false) {}
  int increment() {
    // Comment the following line to see counting fail:
    Guard<FastMutex> g(lock);
    int temp = count ;
    if(rand() % 2 == 0) // Yield half the time
      Thread::yield();
    return (count  = ++temp);
  }
  int value() {
    Guard<FastMutex> g(lock);
    return count;
  }
  void cancel() {
    Guard<FastMutex> g(lock);
    canceled = true;
  }
  bool isCanceled() {
    Guard<FastMutex> g(lock);
    return canceled;
  }
  void pause() {
    Guard<FastMutex> g(lock);
    paused = true;
  }
  bool isPaused() {
    Guard<FastMutex> g(lock);
    return paused;
  }
};
```

```
class Entrance : public Runnable {
  CountedPtr<Count> count;
  CountedPtr<Display> display;
  int number;
  int id;
  bool waitingForCancel;
public:
  Entrance(CountedPtr<Count>& cnt,
    CountedPtr<Display>& disp, int idn)
  : count(cnt), display(disp), number(0), id(idn),
    waitingForCancel(false) {}
  void run() {
    while(!count->isPaused()) {
      ++number;
      {
        ostringstream os;
        os << *this << " Total: "
           << count->increment() << endl;
        display->output(os);
      }
      Thread::sleep(100);
    }
    waitingForCancel = true;
    while(!count->isCanceled()) // Hold here...
      Thread::sleep(100);
    ostringstream os;
    os << "Terminating " << *this << endl;
    display->output(os);
  }
  int getValue() {
    while(count->isPaused() && !waitingForCancel)
      Thread::sleep(100);
    return number;
  }
  friend ostream&
  operator<<(ostream& os, const Entrance& e) {
    return os << "Entrance " << e.id << ": " << e.number;
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  cout << "Press <ENTER> to quit" << endl;
  CountedPtr<Count> count(new Count);
  vector<Entrance*> v;
  CountedPtr<Display> display(new Display);
  const int SZ = 5;
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < SZ; i++) {
      Entrance* task = new Entrance(count, display, i);
      executor.execute(task);
      // Save the pointer to the task:
      v.push_back(task);
    }
    cin.get(); // Wait for user to press <Enter>
    count->pause(); // Causes tasks to stop counting
    int sum = 0;
```

```
    vector<Entrance*>::iterator it = v.begin();
    while(it != v.end()) {
      sum += (*it)->getValue();
      ++it;
    }
    ostringstream os;
    os << "Total: " << count->value() << endl
       << "Sum of Entrances: " << sum << endl;
    display->output(os);
    count->cancel(); // Causes threads to quit
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Count es la clase que conserva el contador principal de los visitantes del jardín. El objeto único Count definido en main() como contador FIXME is held como un CountedPtr en Entrance y, así, se comparte entre todos los objetos Entrance. En este ejemplo, se utiliza un FastMutex llamado lock en vez de un Mutex ordinario ya que un FastMutex usa el mutex nativo del sistema operativo y, por ello, aportará resultados más interesantes.

Se utiliza un Guard con bloqueo en increment() para sincronizar el acceso a count. Esta función usa rand() para realizar una carga de trabajo alta (mediante yield()) FIXME la mitad del tiempo, ...

La clase Entrance también mantiene una variable local numbre con el número de visitantes que han pasado a través de una entrada concreta. Esto proporciona un chequeo doble contra el objeto count para asegurar que el verdadero número de visitantes es el que se está almacenando. Entrance::run() simplemente incrementa number y el objeto count y se duerme durante 100 milisegundos.

En main, se carga un vector<Entrance*> con cada Entrance que se crean. Después de que el usuario pulse Enter, el vector se utiliza para iterar sobre el valor de cada Entrance y calcular el total.

FIXMEThis program goes to quite a bit of extra trouble to shut everything down in a stable fashion.

Toda la comunicación entre los objetos Entrance ocurre a través de un único objeto Count. Cuando el usuario pulsa Enter, main() manda el mensaje pause() a count. Como cada Entrance::run() está vigilando a que el objeto count esté pausado, esto hace que cada Entrance se mueva al estado waitingForCancel, donde no se cuenta más, pero aún sigue vivo. Esto es esencial porque main() debe poder seguir iterando de forma segura sobre los objetos del vector<Entrace*>. Note que debido a que existe una FIXMEpequeña posibilidad que la iteración pueda ocurrir antes de que un Entrance haya terminado de contar y haya ido al estado de waitingForCancel, la función getValue() itera a lo largo de las llamadas a sleep() hasta que el objeto vaya al estado de waitingForCancel. (Esta es una forma, que se conoce como espera activa, y es indeseable. Verá un enfoque más apropiado utilizando wait(), más adelante en el capítulo). Una vez que main() completa una iteración a lo largo del vector<Entrance*>, se manda el mensaje de cancel() al objeto count, y de nuevo todos los objetos Entrance esperan a este cambio de estado. En ese instante, imprimen un mensaje de finalización y salen de run(), por lo que el mecanismo de hilado destruye cada tarea.

Tal y como este programa se ejecuta, verá que la cuenta total y la de cada una de las entradas se muestran a la vez que la gente pasa a través de un FIXMEturnstile. Si comenta el objeto `Guard` en Count::increment(), se dará cuenta que el número total de personas no es el que espera que sea. El número de personas contadas por cada FIXMEturnstile será diferente del valor de count. Tan pronto como haya un `Mutex` para sincronizar el acceso al `Counter`, las cosas funcionarán correctamente. Tenga en cuenta que Count::increment() exagera la situación potencial de fallo que supone utilizar temp y yield(). En problemas reales de hilado, la probabilidad de fallo puede ser estadísticamente menor, por lo que puede caer fácilmente en la trampa de creer que las cosas funcionan correctamente. Tal y como muestra el problema anterior, existen FIXMElikely problemas ocultos que no le han ocurrido, por lo que debe ser excepcionalmente diligente cuando revise código concurrente.

Operaciones atómicas

Note que Count::value() devuelve el valor de count utilizando un objeto `Guard` para la sincronización. Esto ofrece un aspecto interesante porque este código probablemente funcionará bien con la mayoría de los compiladores y sistemas sin sincronización. El motivo es que, en general, una operación simple como devolver un int será una operación atómica, que quiere decir que probablemente se llevará a cabo con una única instrucción de microprocesador y no será interrumpida. (El mecanismo de multihilado no puede parar un hilo en mitad de una instrucción de microprocesador.) Esto es, las operaciones atómicas no son interrumpibles por el mecanismo de hilado y, así, no necesitan ser protegidas.[152] De hecho, si eliminára-mos la asignación de count en temp y quitáramos yield(), y en su lugar simplemente incrementáramos count directamente, probablemente no necesitaríamos un lock ya que la operación de incremento es, normalmente, atómica. [153]

El problema es que el estándar de C++ no garantiza la atomicidad para ninguna de esas operaciones. Sin embargo, pese a que operaciones como devolver un int e incrementar un int son atómicas en la mayoría de las máquinas no hay garantías. Y puesto que no hay garantía, debe asumir lo peor. En algunas ocasiones podría investigar el funcionamiento de la atomicidad para una máquina en particular (normalmente mirando el lenguaje ensamblador) y escribir código basado en esas asunciones. Esto es siempre peligroso y FIXMEill-advised. Es muy fácil que esta información se pierda o esté oculta, y la siguiente persona que venga podría asumir que el código puede ser portado a otra máquina y, por ello, volverse loco siguiendo la pista al FIXMEoccsional glitch provocado por la colisión de hilos.

Por ello, aunque quitar el guarda en Count::value() parezca que funciona no es FIXMEairtight y, así, en algunas máquinas puede ver un comportamiento aberrante.

10.5.3. FIXME:Teminación al bloquear

En el ejemplo anterior, Entrance::run() incluye una llamada a sleep() en el bucle principal. Sabemos que sleep() se despertará eventualmente y que la tarea llegará al principio del bucle donde tiene una oportunidad para salir de ese bucle chequeando el estado isPaused(). Sin embargo, sleep() es simplemente una situación donde un hilo en ejecución se bloquea, y a veces necesita terminar una tarea que está bloqueada.

Un hilo puede estar en alguno de los cuatro estados:

1. Nuevo: Un hilo permanece en este estado solamente de forma momentánea, tan solo cuando se crea. Reserva todos los recursos del sistema necesarios y ejecuta la

inicialización. En este momento se convierte en FIXMEcandidato para recibir tiempo de CPU. A continuación, el planificador llevará a este hilo al estado de ejecución o de bloqueo.

2. Ejecutable: Esto significa que un hilo puede ser ejecutado cuando el mecanismo de fraccionador de tiempo tenga ciclos de CPU disponibles para el hilo. Así, el hilo podría o no ejecutarse en cualquier momento, pero no hay nada que evite FIXME

3. Bloqueado: El hilo pudo ser ejecutado, pero algo lo impidió. (Podría estar esperando a que se complete una operación de entrada/salida, por ejemplo.) Mientras un hilo esté en el estado de bloqueo, el planificador simplemente lo ignorará y no le dará tiempo de CPU. Hasta que un hilo no vuelva a entrar en el estado de ejecución, no ejecutará ninguna operación.

4. FIXMEMuerte: Un hilo en el estado de muerto no será planificable y no recibirá tiempo de CPU. Sus tareas han finalizado, y no será ejecutable nunca más. La forma normal que un hilo tiene para morir es volviendo de su función run().

Un hilo está bloqueado cuando no puede continuar su ejecución. Un hilo puede bloquearse debido a los siguientes motivos:

Puso el hilo a dormir llamando a sleep(milisegundos), en cuyo caso no será ejecutado durante el tiempo especificado.

Suspendió la ejecución del hilo con wait(). No volverá a ser ejecutable hasta que el hilo no obtenga el mensaje signal() o broadcast(). Estudiaremos esto en una sección más adelante.

El hilo está esperando a que una operación de entrada/salida finalice.

El hilo está intentando entrar en un bloque de código controlado por un mutex, y el mutex ha sido ya adquirido por otro hilo.

El problema que tenemos ahora es el siguiente: algunas veces quiere terminar un hilo que está en el estado de bloqueo. Si no puede esperar a que el hilo llegue a un punto en el código donde pueda comprobar el valor del estado y decidir si terminar por sus propios medios, debe forzar a que el hilo salga de su estado de bloqueo.

10.5.4. Interrupción

Tal y como podría imaginar, es mucho más FIXMEmessier salir de forma brusca en mitad de la función Runnable::run() que si espera a que esa función llegue al test de isCanceled() (o a algún otro lugar donde el programador esté preparado para salir de la función). Cuando sale de una tarea bloqueada, podría necesitar destruir objetos y liberar recursos. Debido a esto, salir en mitad de un run() de una tarea, más que otra cosa, consiste en lanzar una excepción, por lo que en ZThreads, las excepciones se utilizan para este tipo de terminación. (Esto roza el límite de un uso inapropiado de las excepciones, porque significa que las utiliza para el control de flujo.)[154] Para volver de esta manera a un buen estado conocido a la hora de terminar una tarea, tenga en cuenta cuidadosamente los caminos de ejecución de su código y libere todo correctamente dentro de los bloques catch. Veremos esta técnica en la presente sección.

Para finalizar un hilo bloqueado, la librería ZThread proporciona la función Thread::interrupted(). Esta configura el estado de interrupción para ese hilo. Un hilo con su estado de interrupción configurado lanzará una Interrupted_Exception si está bloqueado o si espera una operación bloqueante. El estado de interrupción será restaurado cuando se haya lanzado la excepción o si la tarea llama a Thread::interrupted(). Como puede ver, Thread::interrupted() proporciona otra forma de salir de su bucle

run(), sin lanzar una excepción.

A continuación, un ejemplo que ilustra las bases de interrupt():

```
//: C11:Interrupting.cpp
// Interrupting a blocked thread.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class Blocked : public Runnable {
public:
  void run() {
    try {
      Thread::sleep(1000);
      cout << "Waiting for get() in run():";
      cin.get();
    } catch(Interrupted_Exception&) {
      cout << "Caught Interrupted_Exception" << endl;
      // Exit the task
    }
  }
};

int main(int argc, char* argv[]) {
  try {
    Thread t(new Blocked);
    if(argc > 1)
      Thread::sleep(1100);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Puede ver que, además de la inserción de cout, run() tiene dos puntos donde puede ocurrir el bloqueo: la llamada a Thread::sleep(1000) y la llamada a cin.get(). Dando cualquier argumento por línea de comandos al programa, dirá a main() que se duerma lo suficiente para que la tarea finalice su sleep() y llame a cin.get().[155] Si no le da un argumento, el sleep() de main() se ignora. Ahora, la llamada a interrupt() ocurrirá mientras la tarea está dormida, y verá que esto provoca que una Interrupted_Exception se lance. Si le da un argumento por línea de comandos al programa, descubrirá que la tarea no puede ser interrumpida si está bloqueada en la entrada/-salida. Esto es, puede interrumpir cualquier operación bloqueante a excepción de una entrada/salida.[156]

Esto es un poco desconcertante si está creando un hilo que ejecuta entrada/salida porque quiere decir que la entrada/salida tiene posibilidades de bloquear su programa multihilado. El problema es que, de nuevo, C++ no fue diseñado con el sistema de hilos en mente; muy al contrario, FIXMEpresupone que el hilado no existe. Por ello, la librería iostream no ses thread-friendly. Si el nuevo estándar de C++ decide añadir soporte a hilos, la librería iostream podría necesitar ser reconsiderada en el proceso. Bloqueo debido a un mutex.

Si intenta llamar a una función cuyo mutes ha sido adquirido, la tarea que llama será suspendida hasta que el mutex esté accesible. El siguiente ejemplo comprueba si este tipo de bloqueo es interrumpible:

```
//: C11:Interrupting2.cpp
// Interrupting a thread blocked
// with a synchronization guard.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;
using namespace std;

class BlockedMutex {
  Mutex lock;
public:
  BlockedMutex() {
    lock.acquire();
  }
  void f() {
    Guard<Mutex> g(lock);
    // This will never be available
  }
};

class Blocked2 : public Runnable {
  BlockedMutex blocked;
public:
  void run() {
    try {
      cout << "Waiting for f() in BlockedMutex" << endl;
      blocked.f();
    } catch(Interrupted_Exception& e) {
      cerr << e.what() << endl;
      // Exit the task
    }
  }
};

int main(int argc, char* argv[]) {
  try {
    Thread t(new Blocked2);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

La clase `BlockedMutex` tiene un constructor que adquiere su propio objeto `Mutex` y nunca lo libera. Por esa razón, si intenta llamara a f(), siempre será bloqueado porque el `Mutex` no puede ser adquirido. En `Blocked2`, la función run() se parará en la llamada blocked.f(). Cuando ejecute el programa verá que, a diferencia de la llamada a iostream, interrupt() puede salir de una llamada que está bloqueada por un mutex.[157] Comprobación de una una interrupción.

Note que cuando llama a interrupt() sobre un hilo, la única vez que ocurre la interrupción es cuando la tarea entra, o ya está dentro, de una operación bloqueante (a excepción, como ya ha visto, del caso de la entrada/salida, donde simplemente

```
//: C11:Interrupting3.cpp {RunByHand}
// General idiom for interrupting a task.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double PI = 3.14159265358979323846;
const double E = 2.7182818284590452354;

class NeedsCleanup {
  int id;
public:
  NeedsCleanup(int ident) : id(ident) {
    cout << "NeedsCleanup " << id << endl;
  }
  ~NeedsCleanup() {
    cout << "~NeedsCleanup " << id << endl;
  }
};

class Blocked3 : public Runnable {
  volatile double d;
public:
  Blocked3() : d(0.0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        point1:
        NeedsCleanup n1(1);
        cout << "Sleeping" << endl;
        Thread::sleep(1000);
        point2:
        NeedsCleanup n2(2);
        cout << "Calculating" << endl;
        // A time-consuming, non-blocking operation:
        for(int i = 1; i < 100000; i++)
          d = d + (PI + E) / (double)i;
      }
      cout << "Exiting via while() test" << endl;
    } catch(Interrupted_Exception&) {
      cout << "Exiting via Interrupted_Exception" << endl;
    }
  }
};

int main(int argc, char* argv[]) {
  if(argc != 2) {
    cerr << "usage: " << argv[0]
      << " delay-in-milliseconds" << endl;
    exit(1);
  }
  int delay = atoi(argv[1]);
```

```
  try {
    Thread t(new Blocked3);
    Thread::sleep(delay);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

10.6. Cooperación entre hilos

10.6.1. Wait y signal

```
//: C11:WaxOMatic.cpp {RunByHand}
// Basic thread cooperation.
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Car {
  Mutex lock;
  Condition condition;
  bool waxOn;
public:
  Car() : condition(lock), waxOn(false) {}
  void waxed() {
    Guard<Mutex> g(lock);
    waxOn = true; // Ready to buff
    condition.signal();
  }
  void buffed() {
    Guard<Mutex> g(lock);
    waxOn = false; // Ready for another coat of wax
    condition.signal();
  }
  void waitForWaxing() {
    Guard<Mutex> g(lock);
    while(waxOn == false)
      condition.wait();
  }
  void waitForBuffing() {
    Guard<Mutex> g(lock);
    while(waxOn == true)
      condition.wait();
  }
};
```

```
class WaxOn : public Runnable {
  CountedPtr<Car> car;
public:
  WaxOn(CountedPtr<Car>& c) : car(c) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        cout << "Wax On!" << endl;
        Thread::sleep(200);
        car->waxed();
        car->waitForBuffing();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Ending Wax On process" << endl;
  }
};

class WaxOff : public Runnable {
  CountedPtr<Car> car;
public:
  WaxOff(CountedPtr<Car>& c) : car(c) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        car->waitForWaxing();
        cout << "Wax Off!" << endl;
        Thread::sleep(200);
        car->buffed();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Ending Wax Off process" << endl;
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CountedPtr<Car> car(new Car);
    ThreadedExecutor executor;
    executor.execute(new WaxOff(car));
    executor.execute(new WaxOn(car));
    cin.get();
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

10.6.2. Relación de productor/consumidor

```
//: C11:ToastOMatic.cpp {RunByHand}
// Problems with thread cooperation.
//{L} ZThread
#include <iostream>
#include <cstdlib>
```

```
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

// Apply jam to buttered toast:
class Jammer : public Runnable {
  Mutex lock;
  Condition butteredToastReady;
  bool gotButteredToast;
  int jammed;
public:
  Jammer() : butteredToastReady(lock) {
    gotButteredToast = false;
    jammed = 0;
  }
  void moreButteredToastReady() {
    Guard<Mutex> g(lock);
    gotButteredToast = true;
    butteredToastReady.signal();
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        {
          Guard<Mutex> g(lock);
          while(!gotButteredToast)
            butteredToastReady.wait();
          ++jammed;
        }
        cout << "Putting jam on toast " << jammed << endl;
        {
          Guard<Mutex> g(lock);
          gotButteredToast = false;
        }
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Jammer off" << endl;
  }
};

// Apply butter to toast:
class Butterer : public Runnable {
  Mutex lock;
  Condition toastReady;
  CountedPtr<Jammer> jammer;
  bool gotToast;
  int buttered;
public:
  Butterer(CountedPtr<Jammer>& j)
  : toastReady(lock), jammer(j) {
    gotToast = false;
    buttered = 0;
  }
```

```
  void moreToastReady() {
    Guard<Mutex> g(lock);
    gotToast = true;
    toastReady.signal();
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        {
          Guard<Mutex> g(lock);
          while(!gotToast)
            toastReady.wait();
          ++buttered;
        }
        cout << "Buttering toast " << buttered << endl;
        jammer->moreButteredToastReady();
        {
          Guard<Mutex> g(lock);
          gotToast = false;
        }
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Butterer off" << endl;
  }
};

class Toaster : public Runnable {
  CountedPtr<Butterer> butterer;
  int toasted;
public:
  Toaster(CountedPtr<Butterer>& b) : butterer(b) {
    toasted = 0;
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        Thread::sleep(rand()/(RAND_MAX/5)*100);
        // ...
        // Create new toast
        // ...
        cout << "New toast " << ++toasted << endl;
        butterer->moreToastReady();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Toaster off" << endl;
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  try {
    cout << "Press <Return> to quit" << endl;
    CountedPtr<Jammer> jammer(new Jammer);
    CountedPtr<Butterer> butterer(new Butterer(jammer));
    ThreadedExecutor executor;
    executor.execute(new Toaster(butterer));
    executor.execute(butterer);
    executor.execute(jammer);
```

```
    cin.get();
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

10.6.3. Resolución de problemas de hilos mediante colas

```
//: C11:TQueue.h
#ifndef TQUEUE_H
#define TQUEUE_H
#include <deque>
#include "zthread/Thread.h"
#include "zthread/Condition.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

template<class T> class TQueue {
  ZThread::Mutex lock;
  ZThread::Condition cond;
  std::deque<T> data;
public:
  TQueue() : cond(lock) {}
  void put(T item) {
    ZThread::Guard<ZThread::Mutex> g(lock);
    data.push_back(item);
    cond.signal();
  }
  T get() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    while(data.empty())
      cond.wait();
    T returnVal = data.front();
    data.pop_front();
    return returnVal;
  }
};
#endif // TQUEUE_H ///:~
```

```
//: C11:TestTQueue.cpp {RunByHand}
//{L} ZThread
#include <string>
#include <iostream>
#include "TQueue.h"
#include "zthread/Thread.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

class LiftOffRunner : public Runnable {
  TQueue<LiftOff*> rockets;
```

```
public:
  void add(LiftOff* lo) { rockets.put(lo); }
  void run() {
    try {
      while(!Thread::interrupted()) {
        LiftOff* rocket = rockets.get();
        rocket->run();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Exiting LiftOffRunner" << endl;
  }
};

int main() {
  try {
    LiftOffRunner* lor = new LiftOffRunner;
    Thread t(lor);
    for(int i = 0; i < 5; i++)
      lor->add(new LiftOff(10, i));
    cin.get();
    lor->add(new LiftOff(10, 99));
    cin.get();
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

```
//: C11:ToastOMaticMarkII.cpp {RunByHand}
// Solving the problems using TQueues.
//{L} ZThread
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Toast {
  enum Status { DRY, BUTTERED, JAMMED };
  Status status;
  int id;
public:
  Toast(int idn) : status(DRY), id(idn) {}
  #ifdef __DMC__ // Incorrectly requires default
  Toast() { assert(0); } // Should never be called
  #endif
  void butter() { status = BUTTERED; }
  void jam() { status = JAMMED; }
  string getStatus() const {
```

```
    switch(status) {
      case DRY: return "dry";
      case BUTTERED: return "buttered";
      case JAMMED: return "jammed";
      default: return "error";
    }
  }
  int getId() { return id; }
  friend ostream& operator<<(ostream& os, const Toast& t) {
    return os << "Toast " << t.id << ": " << t.getStatus();
  }
};

typedef CountedPtr< TQueue<Toast> > ToastQueue;

class Toaster : public Runnable {
  ToastQueue toastQueue;
  int count;
public:
  Toaster(ToastQueue& tq) : toastQueue(tq), count(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        int delay = rand()/(RAND_MAX/5)*100;
        Thread::sleep(delay);
        // Make toast
        Toast t(count++);
        cout << t << endl;
        // Insert into queue
        toastQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Toaster off" << endl;
  }
};

// Apply butter to toast:
class Butterer : public Runnable {
  ToastQueue dryQueue, butteredQueue;
public:
  Butterer(ToastQueue& dry, ToastQueue& buttered)
  : dryQueue(dry), butteredQueue(buttered) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until next piece of toast is available:
        Toast t = dryQueue->get();
        t.butter();
        cout << t << endl;
        butteredQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Butterer off" << endl;
  }
};

// Apply jam to buttered toast:
class Jammer : public Runnable {
```

```
  ToastQueue butteredQueue, finishedQueue;
public:
  Jammer(ToastQueue& buttered, ToastQueue& finished)
  : butteredQueue(buttered), finishedQueue(finished) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until next piece of toast is available:
        Toast t = butteredQueue->get();
        t.jam();
        cout << t << endl;
        finishedQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Jammer off" << endl;
  }
};

// Consume the toast:
class Eater : public Runnable {
  ToastQueue finishedQueue;
  int counter;
public:
  Eater(ToastQueue& finished)
  : finishedQueue(finished), counter(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until next piece of toast is available:
        Toast t = finishedQueue->get();
        // Verify that the toast is coming in order,
        // and that all pieces are getting jammed:
        if(t.getId() != counter++ ||
           t.getStatus() != "jammed") {
          cout << ">>>> Error: " << t << endl;
          exit(1);
        } else
          cout << "Chomp! " << t << endl;
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Eater off" << endl;
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  try {
    ToastQueue dryQueue(new TQueue<Toast>),
               butteredQueue(new TQueue<Toast>),
               finishedQueue(new TQueue<Toast>);
    cout << "Press <Return> to quit" << endl;
    ThreadedExecutor executor;
    executor.execute(new Toaster(dryQueue));
    executor.execute(new Butterer(dryQueue,butteredQueue));
    executor.execute(
      new Jammer(butteredQueue, finishedQueue));
    executor.execute(new Eater(finishedQueue));
    cin.get();
```

```
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

10.6.4. Broadcast

```
//: C11:CarBuilder.cpp {RunByHand}
// How broadcast() works.
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Car {
  int id;
  bool engine, driveTrain, wheels;
public:
  Car(int idn) : id(idn), engine(false),
  driveTrain(false), wheels(false) {}
  // Empty Car object:
  Car() : id(-1), engine(false),
  driveTrain(false), wheels(false) {}
  // Unsynchronized -- assumes atomic bool operations:
  int getId() { return id; }
  void addEngine() { engine = true; }
  bool engineInstalled() { return engine; }
  void addDriveTrain() { driveTrain = true; }
  bool driveTrainInstalled() { return driveTrain; }
  void addWheels() { wheels = true; }
  bool wheelsInstalled() { return wheels; }
  friend ostream& operator<<(ostream& os, const Car& c) {
    return os << "Car " << c.id << " ["
      << " engine: " << c.engine
      << " driveTrain: " << c.driveTrain
      << " wheels: " << c.wheels << " ]";
  }
};

typedef CountedPtr< TQueue<Car> > CarQueue;

class ChassisBuilder : public Runnable {
  CarQueue carQueue;
  int counter;
public:
  ChassisBuilder(CarQueue& cq) : carQueue(cq),counter(0) {}
  void run() {
```

```
    try {
      while(!Thread::interrupted()) {
        Thread::sleep(1000);
        // Make chassis:
        Car c(counter++);
        cout << c << endl;
        // Insert into queue
        carQueue->put(c);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "ChassisBuilder off" << endl;
  }
};

class Cradle {
  Car c; // Holds current car being worked on
  bool occupied;
  Mutex workLock, readyLock;
  Condition workCondition, readyCondition;
  bool engineBotHired, wheelBotHired, driveTrainBotHired;
public:
  Cradle()
  : workCondition(workLock), readyCondition(readyLock) {
    occupied = false;
    engineBotHired = true;
    wheelBotHired = true;
    driveTrainBotHired = true;
  }
  void insertCar(Car chassis) {
    c = chassis;
    occupied = true;
  }
  Car getCar() { // Can only extract car once
    if(!occupied) {
      cerr << "No Car in Cradle for getCar()" << endl;
      return Car(); // "Null" Car object
    }
    occupied = false;
    return c;
  }
  // Access car while in cradle:
  Car* operator->() { return &c; }
  // Allow robots to offer services to this cradle:
  void offerEngineBotServices() {
    Guard<Mutex> g(workLock);
    while(engineBotHired)
      workCondition.wait();
    engineBotHired = true; // Accept the job
  }
  void offerWheelBotServices() {
    Guard<Mutex> g(workLock);
    while(wheelBotHired)
      workCondition.wait();
    wheelBotHired = true; // Accept the job
  }
  void offerDriveTrainBotServices() {
    Guard<Mutex> g(workLock);
    while(driveTrainBotHired)
```

```
      workCondition.wait();
    driveTrainBotHired = true; // Accept the job
  }
  // Tell waiting robots that work is ready:
  void startWork() {
    Guard<Mutex> g(workLock);
    engineBotHired = false;
    wheelBotHired = false;
    driveTrainBotHired = false;
    workCondition.broadcast();
  }
  // Each robot reports when their job is done:
  void taskFinished() {
    Guard<Mutex> g(readyLock);
    readyCondition.signal();
  }
  // Director waits until all jobs are done:
  void waitUntilWorkFinished() {
    Guard<Mutex> g(readyLock);
    while(!(c.engineInstalled() && c.driveTrainInstalled()
            && c.wheelsInstalled()))
      readyCondition.wait();
  }
};

typedef CountedPtr<Cradle> CradlePtr;

class Director : public Runnable {
  CarQueue chassisQueue, finishingQueue;
  CradlePtr cradle;
public:
  Director(CarQueue& cq, CarQueue& fq, CradlePtr cr)
  : chassisQueue(cq), finishingQueue(fq), cradle(cr) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until chassis is available:
        cradle->insertCar(chassisQueue->get());
        // Notify robots car is ready for work
        cradle->startWork();
        // Wait until work completes
        cradle->waitUntilWorkFinished();
        // Put car into queue for further work
        finishingQueue->put(cradle->getCar());
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Director off" << endl;
  }
};

class EngineRobot : public Runnable {
  CradlePtr cradle;
public:
  EngineRobot(CradlePtr cr) : cradle(cr) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until job is offered/accepted:
```

```
        cradle->offerEngineBotServices();
        cout << "Installing engine" << endl;
        (*cradle)->addEngine();
        cradle->taskFinished();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "EngineRobot off" << endl;
  }
};

class DriveTrainRobot : public Runnable {
  CradlePtr cradle;
public:
  DriveTrainRobot(CradlePtr cr) : cradle(cr) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until job is offered/accepted:
        cradle->offerDriveTrainBotServices();
        cout << "Installing DriveTrain" << endl;
        (*cradle)->addDriveTrain();
        cradle->taskFinished();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "DriveTrainRobot off" << endl;
  }
};

class WheelRobot : public Runnable {
  CradlePtr cradle;
public:
  WheelRobot(CradlePtr cr) : cradle(cr) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until job is offered/accepted:
        cradle->offerWheelBotServices();
        cout << "Installing Wheels" << endl;
        (*cradle)->addWheels();
        cradle->taskFinished();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "WheelRobot off" << endl;
  }
};

class Reporter : public Runnable {
  CarQueue carQueue;
public:
  Reporter(CarQueue& cq) : carQueue(cq) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        cout << carQueue->get() << endl;
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Reporter off" << endl;
  }
```

```
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CarQueue chassisQueue(new TQueue<Car>),
             finishingQueue(new TQueue<Car>);
    CradlePtr cradle(new Cradle);
    ThreadedExecutor assemblyLine;
    assemblyLine.execute(new EngineRobot(cradle));
    assemblyLine.execute(new DriveTrainRobot(cradle));
    assemblyLine.execute(new WheelRobot(cradle));
    assemblyLine.execute(
      new Director(chassisQueue, finishingQueue, cradle));
    assemblyLine.execute(new Reporter(finishingQueue));
    // Start everything running by producing chassis:
    assemblyLine.execute(new ChassisBuilder(chassisQueue));
    cin.get();
    assemblyLine.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

10.7. Bloqueo letal

```
//: C11:DiningPhilosophers.h
// Classes for Dining Philosophers.
#ifndef DININGPHILOSOPHERS_H
#define DININGPHILOSOPHERS_H
#include <string>
#include <iostream>
#include <cstdlib>
#include "zthread/Condition.h"
#include "zthread/Guard.h"
#include "zthread/Mutex.h"
#include "zthread/Thread.h"
#include "Display.h"

class Chopstick {
  ZThread::Mutex lock;
  ZThread::Condition notTaken;
  bool taken;
public:
  Chopstick() : notTaken(lock), taken(false) {}
  void take() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    while(taken)
      notTaken.wait();
    taken = true;
  }
  void drop() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    taken = false;
```

```
    notTaken.signal();
  }
};

class Philosopher : public ZThread::Runnable {
  Chopstick& left;
  Chopstick& right;
  int id;
  int ponderFactor;
  ZThread::CountedPtr<Display> display;
  int randSleepTime() {
    if(ponderFactor == 0) return 0;
    return rand()/(RAND_MAX/ponderFactor) * 250;
  }
  void output(std::string s) {
    std::ostringstream os;
    os << *this << " " << s << std::endl;
    display->output(os);
  }
public:
  Philosopher(Chopstick& l, Chopstick& r,
  ZThread::CountedPtr<Display>& disp, int ident,int ponder)
  : left(l), right(r), id(ident), ponderFactor(ponder),
    display(disp) {}
  virtual void run() {
    try {
      while(!ZThread::Thread::interrupted()) {
        output("thinking");
        ZThread::Thread::sleep(randSleepTime());
        // Hungry
        output("grabbing right");
        right.take();
        output("grabbing left");
        left.take();
        output("eating");
        ZThread::Thread::sleep(randSleepTime());
        right.drop();
        left.drop();
      }
    } catch(ZThread::Synchronization_Exception& e) {
      output(e.what());
    }
  }
  friend std::ostream&
  operator<<(std::ostream& os, const Philosopher& p) {
    return os << "Philosopher " << p.id;
  }
};
#endif // DININGPHILOSOPHERS_H ///:~
```

```
//: C11:DeadlockingDiningPhilosophers.cpp {RunByHand}
// Dining Philosophers with Deadlock.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"
```

```
using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
  srand(time(0)); // Seed the random number generator
  int ponder = argc > 1 ? atoi(argv[1]) : 5;
  cout << "Press <ENTER> to quit" << endl;
  enum { SZ = 5 };
  try {
    CountedPtr<Display> d(new Display);
    ThreadedExecutor executor;
    Chopstick c[SZ];
    for(int i = 0; i < SZ; i++) {
      executor.execute(
        new Philosopher(c[i], c[(i+1) % SZ], d, i,ponder));
    }
    cin.get();
    executor.interrupt();
    executor.wait();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

```
//: C11:FixedDiningPhilosophers.cpp {RunByHand}
// Dining Philosophers without Deadlock.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
  srand(time(0)); // Seed the random number generator
  int ponder = argc > 1 ? atoi(argv[1]) : 5;
  cout << "Press <ENTER> to quit" << endl;
  enum { SZ = 5 };
  try {
    CountedPtr<Display> d(new Display);
    ThreadedExecutor executor;
    Chopstick c[SZ];
    for(int i = 0; i < SZ; i++) {
      if(i < (SZ-1))
        executor.execute(
          new Philosopher(c[i], c[i + 1], d, i, ponder));
      else
        executor.execute(
          new Philosopher(c[0], c[i], d, i, ponder));
    }
    cin.get();
    executor.interrupt();
    executor.wait();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
```

```
} ///:~
```

10.8. Resumen

10.9. Ejercicios